Localized Planning With Diversified Plan Construction Methods

Amy L. Lansky
RECOM Technologies
Artificial Intelligence Research Branch
NASA Ames Research Center
Mail Stop 269-2
Moffett Field, CA 94035-1000

NASA Ames Research Center
Artificial Intelligence Research Branch
Technical Report FIA-93-17
June, 1993

Localized Planning With Diversified Plan Construction Methods

Amy L. Lansky

RECOM/NASA Ames Research Center
Artificial Intelligence Research Branch
MS 269-2, Moffett Field, CA 94035-1000
LANSKY@PTOLEMY.ARC.NASA.GOV

June 18, 1993

Abstract

This paper describes COLLAGE, a planner that utilizes a variety of non-traditional methods for plan construction within a partitioned or *localized* reasoning framework. The COLLAGE domain representation and plan construction methods are based on the use of *action-based constraints*. Not only do such constraints yield highly natural domain encodings, but they are also associated with cost-effective plan construction methods. The additional use of domain structure to partition the reasoning space and guide planning search likewise results in more efficient planning. COLLAGE's unconventional approach to planning is motivated by a specific target domain class: domains with parallel activities that require complex forms of coordination. We describe the design decisions underlying the COLLAGE approach in the context of a novel characterization of planning: a *six-dimensional planning feature space*. We also discuss how this six-dimensional space can serve a larger role in the planning community - as a framework for comparing planning techniques, enhancing communication between researchers, elucidating terminological difficulties, and clarifying the relationship between a domain's characteristics and its most suitable planning technique.

1 Introduction: What Is Planning?

Over the years, the primary focus of the planning community has been on specific planning systems - STRIPS [7], NOAH [30], NONLIN [32], SIPE [35], OPLAN [4], SNLP [27] - to name just a few. Each existing system was built, for the most part, upon the design of its predecessors. As a consequence, most planning systems use the same underlying representations and algorithms - those originally formulated in STRIPS. Indeed, STRIPS-based representation and reasoning has become the defining attribute of AI planning as we know it. Domains are described in terms of state predicates (that describe possible world states) and action descriptions (that define actions in terms of their state-based preconditions and effects). Problem instances are described in terms of initial and goal states. The task of the planner is to come up with an ordered set of actions, all of whose executions are valid with respect to action preconditions and effects, and are guaranteed to transform an initial state into a goal state.

During the 1980's, planning researchers ventured beyond toy problems like the Blocks World into more realistic domains. They quickly bumped up against the inadequacies of this traditional or "classical" formulation of planning. Some researchers developed new methods and representational mechanisms that could be grafted onto a classical planning base. These included techniques for dealing with uncertainty, parallelism, causality, hierarchy and abstraction, metric temporal requirements, and reasoning about resources. Another pivotal development of this period was the work of Chapman [2], who formalized classical planning in terms of the *modal truth criterion* and found it to be NP-complete.

Other planning researchers, inspired by human methods for coping with the world, the harsh requirements of robotic control, and Chapman's analysis, completely abandoned the classical approach for more reactive forms of reasoning. For example, the human tendency to represent knowledge as procedures motivates the architecture of systems like PRS [9] and RAPS [8], which apply user-supplied procedures in reaction to the environment. Similarly, case-based methods [10, 33] are inspired by the human ability to reuse and adapt previously constructed plans in reaction to new problems. The need for quick and flexible response in robotic domains led to architectures based on reactive control rules. Such rules can be constructed either as a byproduct of extensive state-based search [5, 31] or via compilation of user-supplied specifications [11].

In our view, these forays into reactive reasoning served another, perhaps more significant, role in the planning field: they broadened our view of what *constitutes* "planning." A planner is a system that utilizes *any* suitable method for constructing a plan of action. Ideally, one should analyze a particular domain or domain class and determine what planning methods are most suitable for it, from the standpoints of representational ease and plan-construction efficacy.

The most recent work in planning is largely motivated by a desire to better understand this relationship between problem and method. It has emphasized the development of more theoretically well-defined planning frameworks and the use of careful empirical testing. Ironically, however, most of this work has focussed on classical planning (with particular emphasis on SNLP [27]) in rather limited domains. Though the goals and methods of this recent work are laudable, it is unfortunate that these efforts do not examine many of the newer planning methods and ideas - or even some of the more traditional ones (e.g., the use of hierarchical task networks). If analytical and experimental studies are to provide results with utility to real-world domains, they must incorporate newer planning methods as well.

One way to begin a broader form of comparative analysis within the planning community is to organize newer methods and ideas into *dimensions* of the planning process. Each dimension defines a particular aspect of planning, and within each dimension is a set of possible approaches or issues. A specific planner embodies a set of choices made within each planning dimension. The field of "planning" can be defined broadly, yet concretely, as the cross product of these dimensions. If well structured, this kind of categorization can serve as a framework for communication between planning researchers, elucidation of planning terminology, and exploration of different methods for a variety of domains.

Section 2 proposes one such categorization, composed of six dimensions: *(1) domain and problem representation; (2) plan representation; (3) plan construction method; (4) control method; (5) time of plan construction;* and *(6) relationship between planner and environment.* We have found this categorization to be extremely useful in understanding the planning field as a whole and the role of our own non-traditional planner, COLLAGE, in particular.

The rest of the paper describes COLLAGE. We begin in Section 3 by motivating the design choices underlying COLLAGE in terms of the six dimensions. In Section 4 we describe COLLAGE's overall planning approach and its domain and problem representation. This is followed by a description of the COLLAGE plan representation in Section 5 and an in-depth study of the COLLAGE plan construction methods in Sections 6 and 7. Section 8 then describes the localized search control method. It provides both analytical and empirical results that assess the utility of the localized planning approach, including an empirical result that contrasts SIPE's and COLLAGE's performance in an office-building construction domain. Section 8 also discusses the relationship between localization and *abstraction.* Finally, we conclude in Section 9 with a discussion of current research with COLLAGE.

2 Six Dimensions of Planning

This section describes each of the six planning dimensions in terms of a set of "choices" or issues. While not exhaustive, each description includes a variety of techniques that have been used by traditional and non-traditional planners.

1. **Domain and Problem Representation**
   This dimension defines *how domain and problem information is represented.* Among the possibilities are: (1) classical state-based description in terms of state conditions and STRIPS-based action descriptions; (2) action-based description, where domain requirements are described in terms of "behavioral" constraints on the relationships between actions; (3) procedures, defining action sequences that achieve specified goals or effects; and (4) functional input/output requirements. This dimension must also address a variety of semantic issues. For example, can effects depend on whether or not actions occur in parallel? Are actions discrete or continuous? Is probabilistic information or uncertainty about domain state or actions allowed? How does the representation cope with the frame problem and other problems related to scope of effect? Is domain information partitioned or modularized?

2. **Plan Representation**
   This dimension defines *how plans are represented.* Among the possibilities are partially or totally ordered sets of actions, reactive control rules, procedures, code, or neural nets. Other aspects of this dimension include the kinds of relations that can hold between actions, the possible use of metric time-stamping information, the allowance for varying "levels" of plan activity, the modularization of plans into plan fragments, the embedding of constraint networks or other kinds of truth maintenance structures over variables within the plan, and the integration of plan justification structures.

3. **Plan Construction Method**
   This dimension defines *how plans are constructed.* Possibilities include use of traditional algorithms based on the modal truth criterion, methods for combining user-defined procedures or reusing previously generated plans, plan transformation or compilation techniques, problem reduction, chronological projection, action decomposition or goal reduction methods, temporal and causal reasoning over actions and/or states, "CSP-style" constraint propagation on variable bindings, abduction, neural-net reinforcement, and domain-specific methods. Most systems utilize a single method, but some planners allow for mixed methods.

4. **Control Method**
   This dimension defines *how plan construction is controlled* – i.e., the method for controlling the application of plan-construction techniques. Options include search-based, reactive, blackboard-based, or decision-theoretic control, or combinations of these. In some cases, the overall control-space may be partitioned according to various criteria

such as abstraction or localization. Some control schemes are flexible and context-sensitive, while others are rigid.

5. **Time of Plan Construction**
   This dimension deals with *when plan construction methods are applied relative to plan execution* and is thus linked to control. The spectrum of possibilities range from pure advance pre-planning to pure run-time reactive-planning. Intermediate points along this spectrum include reactive systems that are guided by advance reasoning and pre-planning systems that allow for some forms of reactive plan modification.

6. **Relationship Between Planner and Environment**
   This dimension deals with planning autonomy – *how is planning behavior affected by the user and/or environment?* In the past, most AI planners have been completely autonomous. However, some recent planning systems allow for user-input into the planning process. Others are able to learn or modify their methods for plan construction and control based on experience and interaction with the user and/or environment.

Traditional planning systems are quite easy to characterize in terms of the six dimensions. For example, classical systems such as TWEAK [2] or SNLP utilize state-based domain and problem representation and plan construction techniques based on the modal truth criterion. Plans are partially ordered sets of actions and incorporate some limited forms of constraints on variables. Control is search-based with a variety of possible search strategies. Planning is done in advance of execution and is autonomous.

Motivated by the need to cope with more complex domains, the barebones classical framework is extended in systems like NONLIN, SIPE, and OPLAN. Domain representation incorporates the use of hierarchical task networks (i.e. schemas for goal or task reduction) and in some cases, causal theories. Plans incorporate levels of hierarchy as well as certain forms of plan justification structure. Plan construction is expanded to include the use of task reduction and causal reasoning. The search space may be partitioned into abstraction levels. Execution failures may be handled with limited forms of run-time plan modification based on embedded justification information [12]. Finally, the user may be allowed to provide heuristic search guidance.

The ways in which reactive planners diverge from classical planners is best understood by examining the "control," "time", and "autonomy" dimensions. Rather than utilizing an autonomous, search-based, pre-planning approach, reactive systems form plans partially before execution, with the final form of a plan emerging at execution-time. A reactive control mechanism is used to apply pre-constructed plan fragments in response to a dynamically changing environment. Such a strategy is quite appropriate for domains that are highly unpredictable and quickly changing. The various different forms of reactive planning that have been proposed can be distinguished by examining their domain representation, plan representation, and plan construction dimensions, as well as aspects of the autonomy dimension. For example, in systems like PRS [9], a highly procedural domain representation is utilized and procedures are supplied by the user rather than constructed by the system.

The REX/GAPPS approach uses a plan construction method akin to code compilation to generate reactive code fragments. "Universal" planning [31] and systems such as ERE [5] perform advance reasoning, searching a traditionally-based action/state space, the fruits of which are distilled into reactive rules. Case-based methods [10, 33] share much in common with these reactive frameworks in that plan fragments are constructed and later reused. However they differ in the time and control dimensions – reuse may be performed during pre-planning search rather than strictly in reaction to a dynamic environment.

Many common terminological confusions within the classical planning community can be elucidated by the six-dimensional planning view. For example, consider the term "nonlinear." From a plan-representation view, nonlinearity simply means that a plan is a partial rather than a total ordering of actions. If a domain requires parallel forms of activity, the plan representation *must* be "nonlinear." From the standpoint of plan construction, however, "nonlinearity" represents the methodological choice of *least-commitment*; i.e., rather than choosing a particular total ordering, the plan construction method defers that choice by using a partial-ordering.

As another illustration, consider the common confusion surrounding the terms "hierarchy" and "abstraction." The kind of "action-based" hierarchy used by hierarchical task network (HTN) planners denotes a form of task or action decomposition that is explicit in the domain and plan representation and that requires explicit methods during plan construction. In contrast, "state-based" abstractions or hierarchies can be used in planners without HTN capabilities and can be viewed as a mix of domain representation and *control* information. The domain representation is partitioned into levels, each of which incorporates increasing amounts of state-based detail. Planning control is guided by this information; the search space is partitioned into levels, each associated with the state conditions "visible" at that level.

Of course, domain characteristics should provide the ultimate motivation for the choices made in the design of a planner. As new domains emerge, new planning techniques will emerge as well. Consider some of the newest methods that have been proposed – those based on decision theory [34] or neural nets [26]. The six dimensions provide a framework for making planner design choices, and hopefully, for understanding the various tradeoffs between choices. For example, if domain actions have uncertain outcomes, then some form of probabilistic reasoning about actions and their effects should be incorporated. If action outcomes are fairly certain, incorporating such representation and reasoning methods may be a waste of time. Similarly, domains that are fairly stable and require intricate forms of coordination usually require advanced reasoning through a search space; reactive planning would quickly lead to an impasse. On the other hand, extensive search is a waste of time if a domain is changing as you search.

3 Motivation Behind the COLLAGE Planning Approach

COLLAGE[1] [24] is a descendant of the GEMPLAN planner [18, 19, 20, 21]. The design of both systems was motivated by a particular class of target domains: those involving parallel activities that require complex forms of coordination. Most logistical planning domains lie within this class. Indeed, a classic logistical domain, building construction, has been the focus of much of our work. Throughout this paper we will be using a COLLAGE office-building construction domain as the source of several illustrations. Since this same domain was the focus of a study utilizing SIPE [13], it provides an interesting framework in which to contrast our approach with more traditional approaches to planning.

Another logistical domain that has motivated our research is the planning of data selection and data preparation activities performed by Earth scientists. This application is described in more detail elsewhere [23]. It is a good example of a "softbot" domain [6] - i.e., planned actions are executed in the framework of a computer system rather than in the real-world or by a robot. In the data analysis domain, the task of the planner is to select data sets, data transformation algorithms, and execution platforms in a way that meets scientific goals.

We begin our description of COLLAGE with a high-level outline of the COLLAGE architecture, cast in the framework of the six planning dimensions. Within each dimension, we describe the design decisions underlying this architecture, as motivated by the requirements of our target domains. The rest of the paper fills out this description in more detail.

3.1 Domain and Problem Representation

Use of action-based constraints, localized into "scopes of applicability" called **regions**.

Because our focus has been on large domains that require complex forms of coordination, *ease of representation* and *planning efficiency* have been driving forces in the design of COLLAGE. We have found that the requirements of domains in this class can be most clearly described in terms of *direct relationships between actions* rather than in terms of relationships between actions and states. For example, it is has been more natural to use descriptions of the form "A must go before B" than those of form "B has precondition p and A provides condition p."

In COLLAGE, all domain and problem requirements are defined in terms of *action descriptions* and *constraints*. Action descriptions are used to describe the kinds of actions that can be instantiated in a plan. Each description simply provides an action name and a set of parameter types. Constraints are then used to describe the requirements that must be satisfied by the actions in a plan. Each constraint must be an instance of a constraint form in the COLLAGE constraint library.

[1] COordinated Localized aLgorithms for Action Generation and Execution.

In principle, a COLLAGE constraint form can embody *any* kind of requirement. However, all of the current COLLAGE constraint forms are action-based. That is, they describe domain requirements in terms of required relationships between actions and action parameters. We have found that methods for satisfying action-based constraints are, in general, more efficient than algorithms based on the modal truth criterion. In a sense, we have chosen a particular domain and problem representation language that, while general purpose, is more attuned to the requirements of our target domain class. The distinction between action-based domain description and state-based description is discussed at length in Section 7. It is probably the central difference between COLLAGE and traditional planners.

Another important facet of COLLAGE's unorthodox representation is the use of *localization.* We have found that our target domains, though large, have definite structure. In particular, each domain constraint is usually relevant only to a subset of all possible domain activities. This constraint scope or "locality" is usually based on a domain's natural features such as its physical structure or functional processes. Each COLLAGE domain description partitions its associated action descriptions and constraints into localities called *regions.* This partitioning structure is used to semantically define the scope of constraints: each constraint is assumed to be relevant to only the actions within its region and subregions. This semantic information can be viewed as a frame rule. It also serves as a heuristic that partitions and guides the planning process and, as a consequence, usually improves planning efficiency.

3.2 Plan Representation

Partially-ordered set of actions, partitioned into region plans; temporal, causal, simultaneity, and data-flow relationships between actions; use of action hierarchies; embedded "CSP" binding requirements on plan variables; eventually, embedded justification structures.

The COLLAGE plan representation is fairly standard: a plan consists of a set of partially ordered actions. Note that since we are targeting our planner to domains with parallelism, the use of a partial order is *essential* - not just a form of least commitment. There are four kinds of action relationships: temporal, causal, simultaneity, and data-flow. Actions may also be decomposed into interrelated sets of subactions. In contrast to most HTN-based planners, these high-level actions are retained within a plan, even after they are decomposed. This enables reasoning at mixed levels of detail, which can be critical in logistical domains. Plans may also be embedded with a network of binding constraints between plan-variables. In the future, we also intend to incorporate justification structures that will enable flexible forms of dynamic reasoning. Finally, plans are *partitioned* according to domain structure; a full domain plan is composed of region plan fragments.

3.3 Plan Construction Method

Use of a specialized method for each constraint form.

As mentioned above, a COLLAGE domain is represented in terms of constraints, each of which is an instantiation of a constraint form in the COLLAGE constraint library. Each constraint form is associated with a "truth criterion" and constraint satisfaction methods that operationalize this truth criterion. These constraint satisfaction methods construct a plan by adding new actions, relations, and binding requirements into the plan. The COLLAGE architecture is set up in a way that fosters the development of a diverse and easily extendible constraint library - i.e., COLLAGE is intrinsically an integrated framework for mixed-method plan construction. One advantage of using diversified constraints is that it enables flexibility in the way domains can be encoded. Since each constraint is associated with its own plan construction method, a domain's constraint encoding will affect the very nature and cost of the planning process.

3.4 Control Method

Plan-space search, broken up into localized region search spaces; use of agendas to keep track of outstanding plan construction operations.

The control regimen in COLLAGE is plan-space search; each search node is associated with the plan constructed thus far in the planning process . All search is fully backtrackable and the various options and orderings for plan construction steps are processed in a way that ensures a high degree of completeness. Rather than utilizing a single "global" search space, the COLLAGE search space is split up into multiple search spaces, one for each domain region. Each of these regional search spaces is concerned with a subproblem of the overall planning problem - i.e., satisfying a set of region constraints. However, since regions may overlap, these subproblems need not be disjoint - they may be weakly, or even strongly, interacting, depending on the localization structure. The use of localized search is related to the technique of abstraction [22]. It provides many benefits - plan-construction cost reduction, search-space size reduction, and heuristic control - but it also requires a more complicated search mechanism to maintain overall plan consistency and correctness [21, 28].

3.5 Time of Plan Construction

Primarily pre-planning search; currently being extended to allow for "flexi-time" constraint activation, where constraints can be activated at any time - both before and during execution.

Since our domains of interest require complicated forms of coordination, they require extensive pre-planning - complex coordination cannot usually be performed "on-the-fly." Yet, such domains also require various kinds of run-time reasoning as well. In construction

domains, unexpected changes that affect resource availability (tools, contractors, money, time) may necessitate plan modifications. In the data analysis domain, unexpected scientific results may necessitate changes to the data analysis plan. Some constraints should not even be considered until run-time. For this reason, we are currently extending COLLAGE to allow constraint activation and satisfaction at any time relative to execution – what we call *flexi-time* constraint satisfaction.

3.6 Relationship Between Planner and Environment

Autonomous planning, but the user is provided with flexible ways of viewing the planning process; eventually, allow for user input into plan-construction and control.

In general, coordination-intensive domains have always required expert human planners. This is certainly true of our focus domains; consider the general contractor at a building site or scientists selecting and preparing their data. Enabling user input into the planning process can thus provide important planning guidance. It can also help to ensure that an automated planner is eventually accepted for real-world use. We have taken this issue seriously in the design of COLLAGE. Our user interface, COLLIE, allows for sophisticated ways of viewing the planning process. Eventually, we plan on integrating the user more fully into the planning process itself.

4 Planning As Constraint Satisfaction

Central to COLLAGE is the view of planning as "constraint satisfaction." Each domain and problem instance is represented in terms of action-type descriptions, which provide the kinds of actions that can be instantiated, and a set of constraints that must be obeyed by the final plan. Notice that we utilize the term "constraint," not in the confined sense used within the CSP literature[2] [25], but in a much broader sense. In COLLAGE, a constraint is *any kind of property or "truth criterion" that the planner knows how to test and satisfy.* COLLAGE is associated with a broad and easily extendible library of constraint forms. All constraints must be instances of these forms.

Each constraint form in the COLLAGE library is associated with a *check* method, a set of *fix* methods, and a set of *activators* that "operationalize" the formal truth criterion for that form. A check method tests whether a given plan satisfies a constraint. If it does not, the check method returns a set of *bugs* or violation descriptors that describe the ways in which the constraint is violated. Each of these bugs may then be tackled by a *fix* method, which specifies how to augment the plan so that the constraint is satisfied (for that bug). A fix may add new actions, relations between actions, or variable binding requirements into the plan.

[2] A "CSP constraint" is a required relationship between variables that constrains variable assignments.

The role of activators is to indicate whether a constraint is potentially violated. Each activator describes a plan modification (usually, the addition of a certain type of action) that could possibly violate the constraint. The activator set for a constraint form is based on constraint-form semantics and is designed to be conservative – i.e. each constraint should be activated at *least* as often as it needs to be, though perhaps more so. These activators are utilized in a "bottom-up" or "distributed" fashion, activating constraints immediately in response to low-level plan modifications. For example, if a constraint C should be activated by any action of type A, then C will be activated exactly when such an action instance is added into the plan. A constraint is deactivated only after all its bugs have been fixed and if it has not subsequently become reactivated since the last time it was checked.

COLLAGE utilizes the constraints making up a domain and problem description to drive the planning process. Instead of backward- or forward-chaining on goals and conditions, COLLAGE planning is more properly viewed as search through a constraint satisfaction search space (see Figure 1). Each node in the space is associated with a plan constructed up to that point in the search. Upon reaching a node, the planner chooses an activated constraint and outstanding bug for that constraint. It then applies a fix method, yielding a new plan at the next node in the search space. This fix may, as a side effect, activate a new set of constraints.

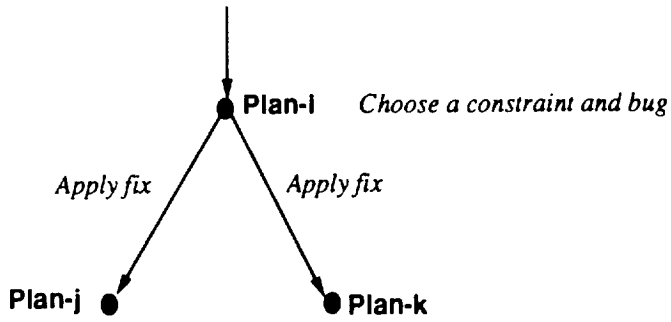


Figure 1: Constraint Satisfaction Search

Figure 1 is actually a simplification of the true COLLAGE search space. In the full space, a node is associated with each choice or branching point in the reasoning process. Among the kinds of choice nodes are the following:

- Selection of particular constraint from a set of activated constraints.
- Selection of a particular bug from a set of constraint bugs that must be tackled.
- Selection of a particular fix method from a set of possible fix methods.

- Choices within a fix method. For example, a fix might choose to instantiate one from a set of possible action types. Or it might choose one from a set of possible relations or binding requirements to insert into the plan.

The COLLAGE search framework maintains a record of all outstanding search choices and allows for fairly liberal orderings between these choices. Because of this flexibility and the ability to backtrack through the choice space, the full planning space will, theoretically, generate nearly all possible plans. However, there are various practical restrictions we have placed on the completeness of this space. A complete description of the COLLAGE search mechanism is provided in Section 8.

Classical planning may be seen as a specialized instance of this constraint-satisfaction view of planning. In traditional frameworks, the only "constraint form" is the achievement of a state-based condition, either at the end of the plan (a goal) or prior to a particular action in the plan (a precondition). All "checks" and "fixes" are based on a single truth criterion – the modal truth criterion – and utilize the usual methods of promotion, demotion, separation, and achievement. All "bugs" are violations of the modal truth criterion – outstanding goals or preconditions that must be established. Thus, rather than searching through a space focussed on constraint-bug violations, a traditional planner chains on goals and preconditions. Finally, the justification structures employed by traditional planners are just one kind of attached plan structure that monitors constraint correctness.

4.1 Domain and Problem Description

In this section we describe the building blocks of the COLLAGE domain and problem representation. Each domain description is composed of a set of *region type definitions*, which collectively define and structure the action type descriptions and constraints of the domain, a set of *regions* that instantiate these region types, and a *domain knowledge base* consisting of domain-specific facts, functions, and type definitions.[3]

$$Domain = < RegionTypes,\ Regions,\ DomainKnowledge >$$

Each region type definition is associated with a unique name and a set of action type descriptions and constraints.

$$RegionType = < Name,\ ActionTypes,\ Constraints >$$

An action type description simply provides a name and set of parameter types.

$$ActionType = < Name,\ ParameterTypes >$$

[3]A COLLAGE problem instance is encoded in terms of constraints and facts. Since these constraints and facts may be associated with *RegionTypes* or *DomainKnowledge*, for purposes of exposition we will take *Domain* to define a full planning problem – both domain requirements *and* a problem instance.

For example, our office building domain includes the following action type descriptions:

```
:action-type (build-column floor coord)
:action-type (build-beam floor coord coord)
```

Each `build-column` action instance represents the act of building a structural column on a particular floor at a specified coordinate location. A `build-beam` action represents the act of building a beam on a particular floor, between two specified coordinates. Both `floor` and `coord` are domain-specific types defined in *DomainKnowledge*. A `floor` parameter may take on values corresponding to building floors. A parameter of type `coord` may take on values corresponding to x-y coordinates in a grid used for defining building locations.

Each constraint C is defined by the name of a constraint form and a set of constraint-parameter values that instantiate that constraint form. C may also be associated with a condition that limits the context in which it is applied, and a set of binding requirements to be imposed upon variables used within C.

$$C = < \mathit{ConstraintFormName}, \mathit{ConstraintFormParameters}, \mathit{Condition}, \mathit{BindingReq} >$$

Notice that each constraint form is associated with its own set of expected constraint-parameters. In Section 6 we provide a full description of each constraint form, including its instantiation parameters, semantic truth criterion, and implementation in terms of checks, fixes, and activators. For now, however, consider the following simple constraints:

```
:constraint
  (tempbefore
     :actions ((build-column ?f ?c1) (build-beam ?f ?c1 ?c2)))
:constraint
  (tempbefore
     :actions ((build-column ?f ?c2) (build-beam ?f ?c1 ?c2)))
```

Each *tempbefore* constraint instance provides two action-type descriptors, $A1$ and $A2$. The constraint requires that each action of type $A2$ be preceded by some action of type $A1$. Thus, the *tempbefore* constraints above require that each instance of a `build-beam` action be preceded by `build-column` actions for each endpoint of the beam.

The portion of *Domain* actually used for plan generation is *Regions* - the region type instances. There may be many instances of each region type; region instances may even be generated dynamically during the planning process. Collectively, a domain's regions contain all action type descriptions and all of the constraints that have to be obeyed by a plan. The structural composition of *Regions* partitions this information in a way that defines the locality or "scope of applicability" of each constraint (see Section 4.2).

Formally, a region R is defined by a unique region name, its region type, and a set of subregions. A region definition may also include a set of region-generator descriptors that are used for generating new subregions during planning. Generated regions are added into *Subregions* dynamically.

$$R = < \textit{RegionName, RT, Subregions, SubregionGenerators} >$$

If *Sub* ϵ *Subregions*, we say that *subregion(R,Sub)* holds. We also use a *descendant* relation between regions, which is the transitive closure of *subregion*. The notation $desc^*(R)$ is used to describe the set of regions composed of R and its descendants. Notice that the *subregion* relation for each COLLAGE domain must form a DAG – no circular relationships between regions can be formed, but regions can be shared by more than one parent. This DAG structure may evolve during planning, as regions are generated. As a simple example, consider the following portion of a very localized version of our office building domain (some constraints have been elided). Figure 2 depicts the *subregion* relation defined by the definitions below.

```
;; REGIONS
(defregion (column-beam-nexus column-beam-nexus-type)
  :subregion all-columns
  :subregion all-beams)

(defregion (all-columns all-columns-type)
  :subregion (:generate (column-builder column-builder-type
                         :limit max-number-of-column-builders)))

(defregion (all-beams all-beams-type)
  :subregion (:generate (beam-builder beam-builder-type
                         :limit max-number-of-beam-builders)))

;; REGION TYPES
(def-region-type column-beam-nexus-type
 :constraint
   (tempbefore
      :actions ((build-column ?f ?c1) (build-beam ?f ?c1 ?c2)))
 :constraint
   (tempbefore
      :actions ((build-column ?f ?c2) (build-beam ?f ?c1 ?c2))))

(def-region-type all-columns-type ...)
(def-region-type all-beams-type ...)

(def-region-type column-builder-type
 :action-type (build-column floor coord))

(def-region-type beam-builder-type
 :action-type (build-beam floor coord coord))
```

This regional structure will apply the two *tempbefore* constraints to all `build-beam` and `build-column` actions associated with generated `beam-builder` and `column-builder` regions. These `beam-builder` and `column-builder` regions may be generated in response

to construction requirements (e.g., requirements encoded as constraints associated with `all-beams` or `all-columns`) and as such may be viewed as "resources" (see Section 7.5).

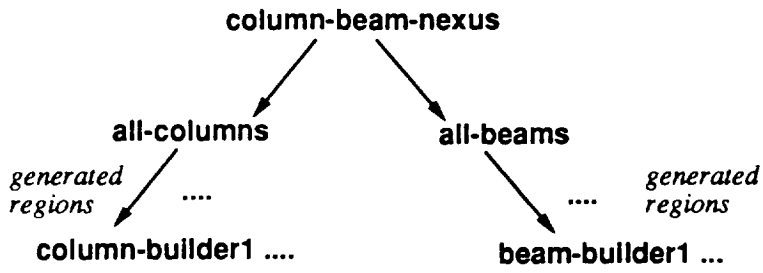


Figure 2: A Simple Region Structure

Finally, *DomainKnowledge* consists of domain-specific facts, functions, and type definitions.

$$DomainKnowledge = < Facts,\ Functions,\ Types >$$

Types includes definitions for parameter types used in *Domain* such as `coord` and `floor`. *Facts* is a set of propositions that describe domain-specific or problem-specific facts. For example, in the data analysis domain, *Facts* will include information about the specific characteristics of Earth projection systems and data-transformation algorithms. *Functions* is a set of domain-specific functions. As we will discuss in Section 6, domain-specific facts and functions can be used to conditionalize the application of domain constraints and to define variable binding requirements. As a result, extensions or modifications of *DomainKnowledge* (say, by scientist users or building contractors) can play an important role in determining the actual course of the planning process.

4.2 Constraint Localization

Given a non-partitioned domain representation consisting of *ActionTypes* and *Constraints*, the task of a planner can be simply defined:

> Find a plan, *Plan*, containing instances of the action types in *ActionTypes*, all of whose possible executions satisfy all constraints in *Constraints*.

In a localized reasoning framework, the planning task is similar, but partitioned:[4]

> For each region R in *Regions*, find a plan, $Plan_R$, containing instances of $ActionsTypes_D$ for any D in $desc^*(R)$, all of whose possible executions satisfy all constraints in $Constraints_R$.

[4]We use the notation X_R to denote information of type X associated with region R.

Intuitively, COLLAGE may be viewed as a set of "mini-planners," each building the portion of the overall plan associated with a particular region, and all linked together as dictated by the *subregion* relationship between regions. COLLAGE creates a reasoning framework for each region R consisting of a planning search tree, $SearchTree_R$, and an agenda, $Agenda_R$, that controls how $SearchTree_R$ is searched. $SearchTree_R$ is concerned with building a plan, $Plan_R$, that satisfies $Constraints_R$. $Plan_R$ may only include actions that are instances of types associated with R or R's descendant regions. These actions form the application "scope" of all constraints in $Constraint_R$.

The details of the COLLAGE plan representation are provided in Section 5. Like the search space, the overall plan is partitioned into region plan fragments. For instance, in the example depicted in Figure 2, the plan for region `column-beam-nexus` will consist of three plan fragments: the plan fragment associated directly with `column-beam-nexus` and the two "subplans" associated with `all-columns` and `all-beams`. These two subplans will then include the plans for generated `column-builder` and `beam-builder` regions.

One consequence of plan partitioning is that region search trees may be jointly constructing *shared* plan fragments. For example, if two regions $R1$ and $R2$ share a common subregion S, $Plan_{R1}$ and $Plan_{R2}$ will share $Plan_S$. One of the tasks of the localized search algorithm is to maintain overall plan consistency in the face of plan partitioning and distribution. This task shares much in common with the task of building and maintaining a distributed, replicated data base. Both tasks must determine criteria for information partitioning and assure consistency maintenance.

As we have already indicated, the localization structure defined by the *subregion* relation defines the semantic "scope of applicability" of regional constraints. Given a localization, COLLAGE will enforce each constraint with respect to its corresponding portion of the plan. As a consequence, the fundamental criterion for choosing a particular domain localization is constraint scope . A valid localization is one that applies each constraint to *at least* all actions relevant to that constraint. Stated more formally, if constraint C is associated with region R and $ActionTypes_C$ is the set of action types that comprise the desired scope of C, then the following must be true:

$$ActionTypes_C \subset \bigcup_{D \in DESC^*(R)} ActionTypes_D$$

Notice that a valid localization *may* allow constraints to be applied to actions outside their "true" scope. Expanding the scope of applicability of a constraint may be motivated by a desire to reduce regional sharing and consistency maintenance costs. For example, in the scenario described above, if $Plan_S$ forms the bulk of both $Plan_{R1}$ and $Plan_{R2}$, it may actually be more cost-effective to collapse regions $R1$, $R2$, and S together to form a single region consisting of all of their action types and constraints. Such a collapse will not cause a lack of correctness – it will simply cause some constraints to be tested with respect to irrelevant portions of the plan.

In practice, the localization structure chosen for a domain will be strongly influenced by domain-dependent features such as its physical structure, its functional components, its

agents or processes, its temporal compartments (e.g., each week of a schedule), or levels of abstraction. In the office building domain, a mix of criteria is used: the scope of individual constraints, the physical partitioning of the building into floors and rooms, the contractor "agents," resources (e.g., tools), task types (e.g., flooring, ceiling preparation), and levels of detail or abstraction. All of these kinds of partitioning are superimposed within a single domain description.

Localization has several benefits and some pitfalls, many of which are discussed in greater detail in Section 8.3. In general, a good localization must strike a balance between increased partitioning (fine-tuning the applicability of constraints) and the increased consistency maintenance costs that result from region-sharing and interaction [21]. One of our research goals is to understand this tradeoff more deeply.

5 Plan Representation

Each COLLAGE region plan is a complex data structure consisting of several types of information. In this section we describe how plans are partitioned according to region structure, as well as the various kinds of information that can be found in a plan.

5.1 Plan Structure

Given a region R with subregions $S1$,... Sn (which may be static or dynamically generated), $Plan_R$ will have the following form:

$$Plan_R = < LocalPlan_R, Plans_{S1}, \ldots Plans_{Sn} >$$

A plan for a region R includes a local plan, $LocalPlan_R$, consisting of information associated directly with R, and plans for each of its subregions. Note that $Plans_{S1}$,.... $Plans_{Sn}$ will include plans for each of their subregions. Thus, $Plan_R$ can be viewed as a set of local plans, one for R and one for each of R's descendant regions.

All plan information is stored in the local plan of some region. When a fix needs to add new information into a plan (e.g., an action, relation, or binding requirement), COLLAGE must decide which region's local plan to associate this information with. This decision is critical, since *where* information is stored will determine which regions have access to it. When reasoning within the framework of region R, COLLAGE will store any piece of plan information within the local plan of the "lowest" or "smallest" region (or, in some cases, regions) within R that can contain the information. This strategy enables all plan information to be visible to all relevant regions. One corollary of this strategy is that each action instance is stored in the local plan of the region that contains that action's type definition.

As an example, consider the localization depicted in Figure 3, consisting of five regions *R1,...R5*. Suppose that a fix method associated with a constraint in *R*1 needs to add four new actions (`a, b, c, d`) and three new temporal relations ((`a => b`), (`b => c`), (`c => d`)) into $Plan_{R1}$. Since actions `a` and `b` are instances of action types in region *R*4, they are associated with $LocalPlan_{R4}$. Similarly, `c` is associated with $LocalPlan_{R5}$ and `d` is associated with $LocalPlan_{R3}$. Given this localization structure, the relation (`a => b`) will also be associated with $LocalPlan_{R4}$. However, since *R*2 is the nearest common ancestor of *R*4 and *R*5, (`b => c`) will be associated with $LocalPlan_{R2}$. Similarly, (`c => d`) will be associated with $LocalPlan_{R1}$.

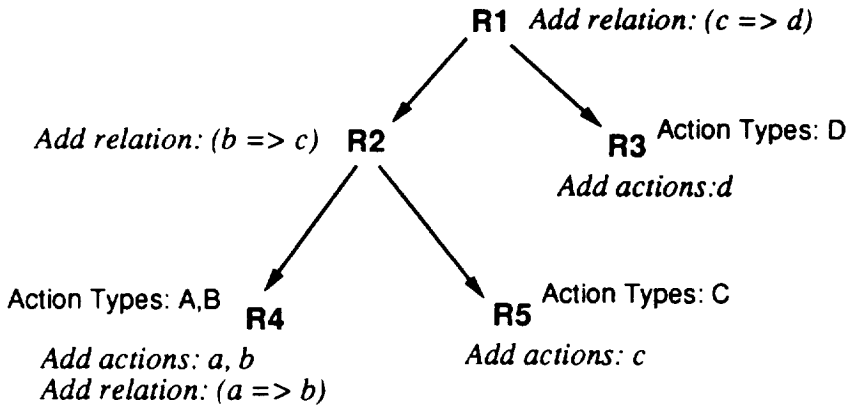


Figure 3: Plan Structure

This strategy of storing plan information as "locally" as possible also increases the need for consistency maintenance. For example, if *R*4 is a descendant of some other region *R*6 not depicted in the figure, the plan information stored in $LocalPlan_{R4}$ will also be part of $Plan_{R6}$. Thus, reasoning within *R*1 may result in the addition of information to $LocalPlan_{R4}$ and, ultimately, that change will have to propagate to $Plan_{R6}$ and to any other plan that contains $LocalPlan_{R4}$. This process is described in further detail in Section 8.

5.2 Plan Content

Each local plan in COLLAGE may contain several different kinds of information: actions, relations between actions, and binding requirements on action parameters. This section describes this plan content in more detail.

5.2.1 Actions

The most basic piece of plan information is an action instance. Each action is a unique object consisting of a name and a set of typed parameter objects. A parameter object is either simple or structured (i.e., composed of subsidiary parameter objects called slots). For instance, the `floor` parameter type is a simple type; `coord` is a structured type composed of `x` and `y` coordinate slots. Each simple parameter object must be either a constant value or a *variable* object. Each variable object is associated with a set of possible values.

For example, consider an action instance `(build-column 1 {2,2})`. This action represents the act of building a column for floor 1 at coordinate {2,2}. Another action instance might be `(build-column F {1,2})`. This represents the act of building a column at coordinate {1,2}. However, the floor parameter variable object `F` may not yet be bound to a distinct value.

COLLAGE actions may be *atomic* or *nonatomic*. Ontologically, atomic actions represent discrete instants or points in time. A nonatomic action is composed of subactions, which may be either atomic or nonatomic. Each nonatomic action must be associated with at least one distinguished "first" subaction and at least one distinguished "last" subaction. These "endpoints" can be used to establish interval-based relationships between nonatomic actions [19].

5.2.2 Action Decomposition Information

When a nonatomic action is decomposed into subactions (via the application of a *decompose* constraint – see Section 6), information about action/subaction relationships must be recorded. Rather that removing a nonatomic action from a plan and replacing it with its subactions (as is done by most HTN-based planners), COLLAGE retains all nonatomic actions within a plan and additionally stores information about the hierarchical relationship between actions and their subactions. In particular, two types of information are stored:

1. Relations of form `(subaction a sub)`.
2. Relations of form `(firstsubaction a sub)` and `(lastsubaction a sub)`.

By retaining actions at all levels of detail within the plan, constraints can be applied to plans that incorporate actions at mixed-levels of detail. For instance, in building construction domains, high-level plumbing activities may be temporally constrained relative to low-level

electrical activities. Often, action/subaction boundaries are used as criteria for abstraction-based localization structures. The relationship between localization and abstraction is discussed in more detail in Section 8.4.

5.2.3 Relations between actions

Besides decomposition relations, there are four other types of relations that can exist between actions:[5]

- **Temporal Relations**
  The presence of a relation (`a1 => a2`) between two actions `a1` and `a2` indicates that `a1` must occur before `a2`.[6] The temporal relation is transitive and is also propagated as a result of action decomposition. (A complete description of the temporal relations implied by action decomposition is described in Section 6.)

  COLLAGE maintains full, explicit temporal closure within each region plan and performs this closure incrementally. Maintaining a temporally closed plan, while costly, can greatly improve the efficiency of plan construction algorithms, reducing temporal inference costs to near constant-time look-up. Our experience with COLLAGE has shown that the price for temporal closure maintenance is mitigated by the fact that this closure is only performed on a regional basis rather than with respect to a full "global" plan. These locally closed regional "islands" are similar to the reference intervals described by Allen [1].

- **Causal Relations**
  The relation (`a1 ~> a2`) indicates that `a1` causes or enables `a2`. In our ontology, causality implies temporal precedence: (`a1 ~> a2`) $\supset$ (`a1 => a2`). This implicant is explicitly stored in a COLLAGE plan. However the causal relation is *not* transitive. As a consequence, the causal relation provides a convenient mechanism for representing one-to-one relationships between actions.

  For example, the act of opening a door may be viewed as enabling the act of walking through the door: (`open-door ~> enter-room`). Though many other `open-door` actions may temporally precede a particular `enter-room` action, domain requirements may be defined so as to ensure that only one `open-door` action actually enables each `enter-room` action.

[5] For the sake of clarity, we use infix notation throughout this paper for the temporal, causal, simultaneity, and dataflow relations. However, in COLLAGE itself, these relations are stored using a more traditional prefix form - e.g., (`before a b`) rather than (`a => b`).

[6] Although our logic of actions is point-based, notice that the first and last subactions of nonatomic actions can be interrelated by the temporal and simultaneity relations so as to emulate all of Allen's interval relations between actions [1, 19].

- **Simultaneity Relations**
  A relation (a1 ⇌a2) indicates that a1 and a2 must occur at the same time in all possible executions of a plan. Currently, only atomic actions can be related by the simultaneity relation. However, two nonatomic actions could be viewed as "simultaneous" if their respective first and last subactions are simultaneous. Notice that any two actions that are unrelated temporally can *potentially* occur simultaneously in some execution of a plan. However, if they are related by ⇌, they *must* occur simultaneously in all executions.[7]

- **Data Flow Relations**
  The dataflow relation >> was recently incorporated into COLLAGE to facilitate description of certain kinds of behavior in the data analysis domain. Dataflow implies temporal flow: (a1 >> a2) ⊃ (a1 => a2). It also implies a relationship between specific parameters of a1 and a2 - those of type pipe. A pipe parameter represents a data flow "pipe." It is a structured parameter object consisting of pipe identifier, input value, and output value slots. If (a1 >> a2) holds and a1 and a2 have pipe parameters with the same pipe identifier, COLLAGE will insure that the pipe output value associated with a1 is the same as the pipe input value associated with a2.

5.2.4 Binding requirements on action parameters

The data flow relation represents one specialized way of imposing a constrained relationship between action parameters. COLLAGE also allows "CSP-style" binding requirements between action parameters to be embedded within a plan. These binding requirements may be imposed either due to the explicit binding requirements associated with a constraint or as a byproduct of a fix. For example, if the constraint

```
:constraint
  (tempbefore
     :actions ((build-column ?f ?c1) (build-beam ?f ?c1 ?c2)))
```

is applied to a plan containing the two action instances

```
(build-column F {1,1})
(build-beam 1 {1,1} {1,2})
```

and floor parameter variable F is unbound, the constraint fix method may add a temporal relation between the two actions as well as the binding requirement (= F 1).

[7]For example, in COLLAGE, the partition order defined by ((a => b),(a => c)) has three possible executions: {a,b,c}, {a,c,b}, and {a,bc}. However, if we add the additional relation (b =c) into the partial order, the last of these three executions is the only possible execution. A complete formal semantics of action-based representation and plan execution is provided in [18, 19], which is based on a first-order temporal logic of actions. This logic was originally formulated for specifying and verifying concurrent programs [17] and later provided the formal basis for both GEMPLAN and COLLAGE. However, in this paper, we use a non-modal logic for describing the truth criteria of the COLLAGE constraint forms.

Currently, all binding requirements must be unary or binary relations between *elementary binding objects*. An elementary binding object is either:

- A constant.
- A constant-valued parameter object.
- A variable parameter object of some enumerable type.

Each binding requirement may be defined by a boolean-valued Lisp function, a boolean-valued function defined in *DomainKnowledge*, or by facts in *DomainKnowledge*. For example, suppose that we have facts `(hard-flooring wood)`, `(hard-flooring vinyl)`, and `(hard-flooring tile)`, and a boolean-valued function `(color-match color1 color2)`. Further suppose that we have actions of type `(lay-flooring floortype floorcolor)` in our plan. Using the `hard-flooring` facts, we could impose a unary binding requirement on `floortype` parameters. Using `color-match`, we could require that if two rooms abut, their two `floorcolor` parameters must be related by `color-match`.

Note that, like all plan information, binding requirements are stored within local plans. The binding requirements for each region plan are grouped together in a network, much in the style of CSP-networks [25]. However, rather than using a single network for the entire plan, a set of sub-nets are formed, each consisting of binding requirements imposed on the action parameters within a particular region. The consistency of the entire "global" network is maintained much the same way as overall plan consistency is maintained, and is described further in Section 8.

6 Plan Construction Methods

6.1 Constraint Application Semantics

We begin our description of the COLLAGE constraint forms by describing the overall semantics of constraint activation and the application of checks and fixes. COLLAGE takes the *ConstraintFormName* and *ConstraintFormParameters* for each constraint C and yields a check method $Check_C$, fix methods $Fix1_C$ ... $FixN_C$, and activators $Activators_C$ that operationalize the semantic truth criterion for C. Each C is also associated with an initial activation setting $InitialActivationSetting_C$, which has value `ON` or `OFF`, indicating whether C is considered to be active or inactive when planning begins.

As discussed in Section 4.1, each constraint C may also be associated with a condition, $Condition_C$, and a set of binding requirements, $BindingReq_C$. A constraint C need be satisfied only if $Condition_C$ is satisfied by the plan. $BindingReq_C$ imposes further requirements on the bindings of action parameters referenced within C. The treatment of binding requirements, constraint conditions, and the relationship between conditions and constraint activation are discussed at length in Section 6.3. In the following discussion, we assume that both

$Condition_C$ and $BindingReq_C$ are empty - i.e. $Condition_C$ is assumed to be satisfied and there are no additional binding requirements.

A check method takes a plan as input and returns a set of bug descriptors for that constraint. If $Check_C(Plan)$ returns an empty set of bugs, we know that C is satisfied by *Plan*; i.e., $Plan \models TruthCriterion_C$. Each bug descriptor describes a particular way in which C is violated by *Plan*. A fix method takes a plan and a bug descriptor as input and, if successfully executed, returns a newly "fixed" plan.

A side-effect of executing some $FixK_C$ may be the activation of other constraints, including, possibly, C itself. This is because the new actions, relations, and binding information added into a plan by $FixK_C$ may potentially violate the truth criteria of those constraints, introducing new bugs. The activators for each constraint C are designed to be conservative. If constraint C *is* violated by some plan modification, C ***will*** be activated by some activator in $Activators_C$. However, some activators may be overly conservative - they may occasionally activate C unnecessarily. The maintenance of this constraint activation information is discussed in detail in Section 8.

Given our definition of constraint check, fix, and activation semantics, we have the following observation:

Given constraint C and plan $Plan_0$, if the following holds:

- $\{Bug_1 ... Bug_m\} \leftarrow Check_C(Plan_0)$
- A sequence of fix methods are successfully applied, one for each bug in the set:

  $(\forall\ i:\ 1..m)\ Fix^i_C \in \{Fix1_C ... FixN_C\}$

  $Plan_1 \leftarrow Fix^1_C(Plan_0, Bug_1)$

  :

  $Plan_m \leftarrow Fix^m_C(Plan_{m-1}, Bug_m)$

- C is *not* activated by any Fix^i_C

THEN $Check_C(Plan_m)$ must return no bugs - i.e. $Plan_m \models C$.

In Section 6.2 we provide an in-depth description of the current COLLAGE constraint library. For each constraint form, we provide a truth criterion, an algorithmic description of the constraint check and fix methods, the constraint activators, an initial activation setting, and a brief discussion of constraint complexity. We also provide examples of constraint use. First, however, we must discuss some issues relevant to our description of these constraint forms.

6.1.1 Action Descriptors

The various constraint parameters that are used to instantiate constraint forms are all composed from *action descriptors*. In this paper, we also use action descriptors to describe the truth criterion for each constraint form. An action descriptor is similar to an action-type description. It provides a skeletal description that can be used to match against and retrieve action instances in a plan. All COLLAGE plan information is stored and indexed in such a way that facilitates quick matching with these action descriptors. *Relation descriptors*, which are composed of a relation name and two action descriptors, can also be used to quickly retrieve matching plan relations.

Each action descriptor A has the form:

$$A = < Name, ParameterDescriptors >$$

Name is the simple token name of an action type. Each parameter descriptor is either a constant value or a variable descriptor ?v. An action instance *necessarily matches* an action descriptor if they have the same name and if their corresponding parameter objects and descriptors necessarily match. Similarly, an action *possibly matches* an action descriptor if they have the same name and their corresponding parameter objects possibly match. Figure 4 depicts the conditions under which parameter objects and parameter descriptors possibly or necessarily match.

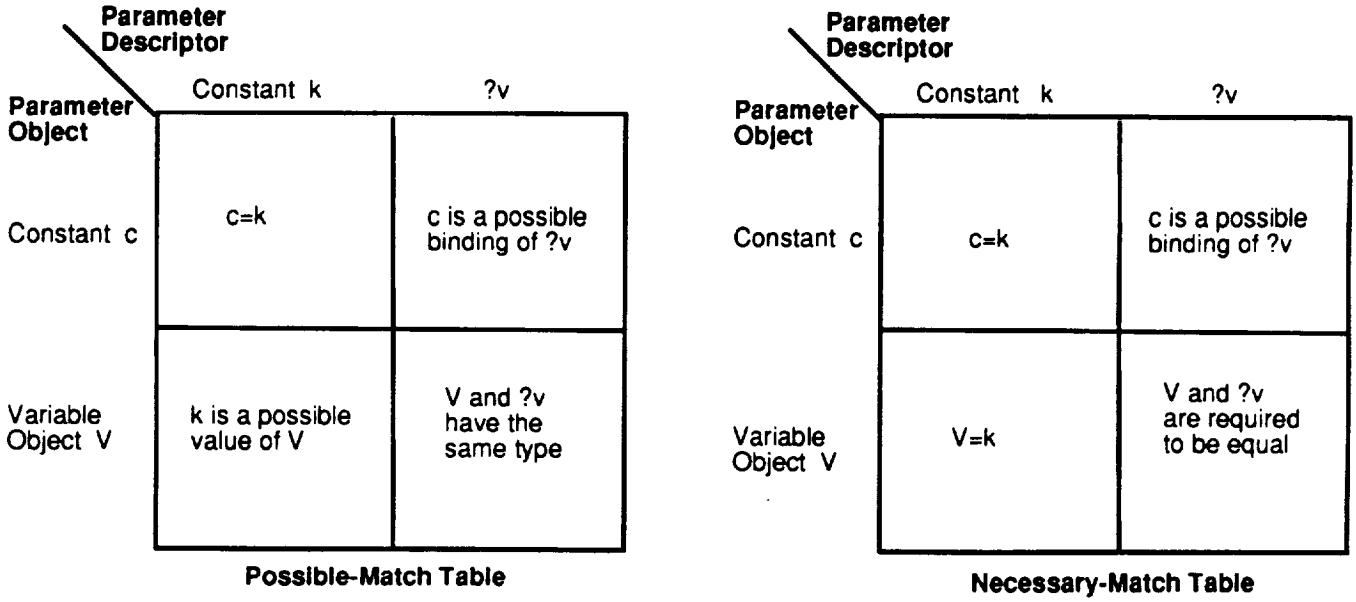


Figure 4: Matching Tables

For example, an action instance (`build-column 1 {0,1}`) necessarily matches the descriptor (`build-column 1 {?x ?y }`) if `0` is a valid binding of `?x` and `1` is a valid binding of `?y`. However, (`build-column F {0,1}`) only *possibly* matches this descriptor if `F` has not yet been bound. It does *not* match if `F` has been bound to 2.

During constraint checking and fixing, parameter descriptors of form `?v` used within C will be matched with parameter objects associated with actions in a plan. Each application of a check or fix algorithm is associated with a "current binding list" for these parameter descriptors. This binding list is passed through the execution of a check, attached to outgoing bugs, and later passed into the execution of a fix. The possible binding for each of these descriptors is further constrained as it is matched with more parameter objects. Ultimately, COLLAGE assures that all parameter objects matched against the same parameter descriptor take on the same value. This mechanism is further described in Section 6.3.

6.1.2 Plan Inheritance

As described earlier, each region plan, $Plan_R$, is composed of a local plan for R and the local plans of R's descendant regions. Rather than making a new copy of a complete R plan each time that plan is modified, COLLAGE stores plan changes in an incremental fashion, associating new plan changes with new local plans. Each local plan inherits plan content from predecessor local plans. A full region plan inherits the content of all its local plans. While this scheme saves on storage costs, the cost of looking up plan information is increased by the need to follow the local plan inheritance chain.

For instance, suppose that region R has a subregion S and S has no subregions (see Figure 5). Let us assume that a region R fix is given an input plan consisting of $LocalPlan_{R_i}$ and $LocalPlans_{S_j}$ and must add plan information to both local plans. First it will create new local plans for both regions, $LocalPlan_{R_{i+1}}$ and $LocalPlans_{S_{j+1}}$. These new plans will be linked into the plan-inheritance structure so that $LocalPlan_{R_{i+1}}$ inherits $LocalPlan_{R_i}$ and $LocalPlans_{S_{j+1}}$ inherits $LocalPlans_{S_j}$. The fix will then add the new plan information into the new local plans, yielding a new plan for R consisting of $LocalPlan_{R_{i+1}}$ and $LocalPlans_{S_{j+1}}$ (as well as a new plan for S consisting of $LocalPlans_{S_{j+1}}$) and the local plans they inherit.

6.1.3 Choice Points

Each of the fix methods described in Section 6.2 may include *choice points*: points at which choices must be made during the course of executing a fix. Like other higher level choices in the search process, each internal fix choice is associated with a search node. When search backtracks to a choice point, the other possible choices at that node can be pursued. This backtracking will occur in response to failures of fix steps; in particular, if a fix step fails, search will backtrack directly to the most recent choice node. All of the choices within a fix will be fully explored (via backtracking) before an entire fix fails and is abandoned.

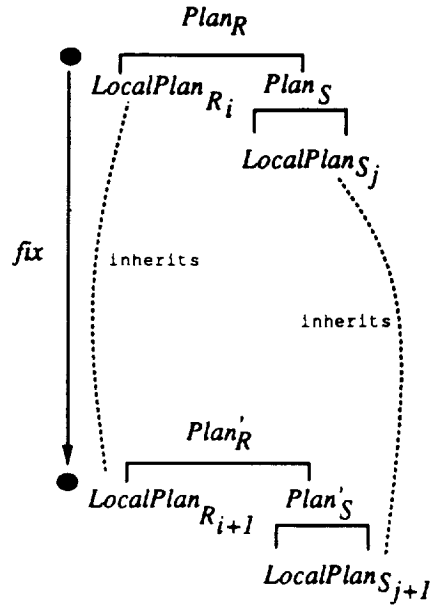


Figure 5: Plan Inheritance

Note that many kinds of fix steps can fail – e.g., a plan-content query or the addition of a relation (if that relation introduces temporal inconsistency into the plan). In our description of the fix algorithms, most of these possible points of failure are left implicit rather than explicitly noted.

6.1.4 Complexity Measures

The complexity measures provided for each constraint form described in Section 6.2 are based on several assumptions. First, each assertion to the plan is assumed to take constant time. Thus, we do not take into account the cost of incremental temporal closure and binding constraint propagation that must be performed in response to new plan relations and bindings. Though these "hidden" costs of closure maintenance and binding propagation are not reflected in the constraint cost measures, a separate description of these mechanisms and their complexity is provided in Section 6.4.

We also make a second assumption – that each plan look-up takes time t_{look}. COLLAGE utilizes a discrimination tree to store plan information. This tree enables near constant-time retrieval of matching actions and relations. However, this retrieval is not plan-relative – i.e., matching actions and relations for *all* local plans are returned. Thus, the cost t_{look} for a *particular* plan depends on the cost of additionally checking the plan-inheritance chain for that plan.

6.2 Constraint Form Library

6.2.1 Action Constraint Form: Action({A1...An})

The most simple constraint form in COLLAGE is the *action* constraint. Action constraints are used to ensure that particular actions are present in the plan. As such, they are usually used to express problem-specific goals. Each action constraint provides a set of action descriptors *A1...An*. The constraint is satisfied if, for each action descriptor *Ai*, there exists some action in the plan that necessarily matches *Ai*.[8]

Truth Criterion: $\forall$ *Ai* ϵ {*A1...An*} ($\exists$ *ai:Ai*)

Activators: none

Initial Activation Setting: ON

Check Algorithm

Check(Plan)

For each *Ai* in *A1...An*
    If there is no action that necessarily matches *Ai* then
        Add *missing-action(Ai)* to *BugSet*
Return *BugSet*

Fix Algorithm[9]

Fix(missing-action(Ai),Plan)

Create an action instance *ai* that necessarily matches *Ai*
Add *ai* into *Plan'*
Return *Plan'*

The complexity of the check algorithm is $O(nt_{look})$, where n is the number of actions that match the action descriptor set. The fix for each bug takes constant time. Notice that since the initial activation setting is on for this constraint form, all action constraints are

[8] We use the notation *a:A* to denote that action *a* necessarily matches descriptor *A*. The notation *a:{A1...An}* denotes an action *a* that necessarily matches one of the descriptors {*A1...An*}. Also note that quantification is assumed to be nested. For example, given *($\forall$ a:A)($\forall$ b:B) expr(a,b)*, if *A* and *B* contain parameter descriptors in common, all parameter objects for action instance pairs *(a,b)* will necessarily match as required.

[9] We use the notational convention that *Plan'* is the output plan for each fix. This *Plan'* is an augmentation of the input plan *Plan*.

active when planning begins. Moreover, since the activator set is empty, an action constraint cannot be violated once it is satisfied.

As stated above, action constraints are usually used for expressing goals. Often, they are used in concert with decompose constraints; an action constraint might add a high-level nonatomic action into the plan that is later decomposed into lower-level subactions. For example,

```
:constraint
 (action
   :actions ((do-flooring 1)
             (do-flooring 2)))
```

would result in the addition of high-level flooring actions for floors 1 and 2 that would later be decomposed into lower-level flooring actions. However, there is actually a more elegant way to instantiate problem-specific goals like these – via constraint conditionalization. For example, the following constraint will ensure that `do-flooring` actions for all floors above floor 0 are added to the plan. Facts of form `(floor 1)`, `(floor 2)`, etc. in *DomainKnowledge* are used to describe the floors and other features of a particular office building.

```
:constraint
  (action
    :condition ((fact (floor ?f))
                (test (> ?f 0)))
    :actions ((do-flooring ?f)))
```

The semantics of constraint conditionalization is described fully in Section 6.3.

6.2.2 Temporal/Causal Constraint Forms

This class of constraints consists of four binary constraint forms: *tempbefore, tempafter, enable,* and *cause.* Each temporal/causal constraint provides two action type descriptors, A and B. The constraint is satisfied if matching action instances of type A and B exist in the plan which are temporally or causally related in a specified way. Below, we provide the truth criteria for all four constraint forms, and the check and fix algorithms for *tempbefore.* The checks and fixes for the other three forms are similar.

Truth Criteria:

Tempbefore(A B): *(∀ b:B) (∃ a:A) (a => b)*
Tempafter(A B): *(∀ a:A) (∃ b:B) (a => b)*
Enable(A B): *(∀ b:B) (∃ a:A) (a ~> b)*
Cause(A B): *(∀ a:A) (∃ b:B) (a ~> b)*

Activators:

Tempbefore(A B): $\{B\}$
Tempafter(A B): $\{A\}$
Enable(A B): $\{B\}$
Cause (A B): $\{A\}$

Initial Activation Setting: `OFF`

Tempbefore Check Algorithm

Check(Plan)

For each action *b* possibly matching *B*
    If there is no relation `(a => b)`, where *a* necessarily matches *A*, then
        Add *missing-predecessor(b)* to *BugSet*
Return *BugSet*

Tempbefore Fix Algorithm 1: Using existing action

Fix(missing-predecessor(b),Plan)

Find action instances *a1...am* that possibly match *A*
CHOICE POINT: Choose an *ai* from *a1...am*
Add binding relations into *Plan'* so that *ai* necessarily matches *A*
Add `(ai => b)` into *Plan'*
Return *Plan'*

Tempbefore Fix Algorithm 2: Create new action

Fix(missing-predecessor(b),Plan)

Create an action instance *a* that necessarily matches *A*
Add *a* and `(a => b)` into *Plan'*
Return *Plan'*

The complexity of the check algorithm is $O(nt_{look})$ where n is the number of actions that match A or B. The first fix algorithm is $O(nt_{look})$, where n is the number of actions that match A. The second fix algorithm is constant time. Notice that backtracking over the choice point in the first fix algorithm can yield m possible solutions for each bug. Two examples of *tempbefore* constraints were provided in Section 4.1.

6.2.3 All-Matching Constraint Forms

This class of constraints is also composed of four constraint forms: *all-matching-before, all-matching-after, all-matching-enable*, and *all-matching-cause*. They are used to ensure that all actions that match a particular action descriptor bear a specified causal or temporal relationship with respect to each action of another specified form. Unlike the temporal/causal constraint forms, the fix methods for these constraint forms will not add new actions into a plan – they will only create new relations between existing actions. The truth criteria for all four constraint forms are provided below. The check and fixes for the *all-matching-tempbefore* constraint are also provided; checks and fixes for the other three constraint forms are similar.

Truth Criteria:

All-Matching-Before(A B): *(∀ b:B) (∀ a:A) (a => b)*

All-Matching-After(A B): *(∀ a:A) (∀ b:B) (a => b)*

All-Matching-Enable(A B): *(∀ b:B) (∀ a:A) (a ~> b)*

All-Matching-Cause(A B): *(∀ a:A) (∀ b:B) (a ~> b)*

Activators: {*A*,*B*}

Initial Activation Setting: `OFF`

All-Matching-Before Check Algorithm

Check(Plan)

For each action *b* possibly matching *B*
  For each action *a* necessarily matching *A*
    If there is no relation `(a => b)` then
      Add *missing-relation(a b)* to *BugSet*
Return *BugSet*

All-Matching-Before Fix Algorithm

Fix(missing-relation(a b),Plan)

Add `(a => b)` into *Plan'*
Return *Plan'*

The complexity of the check algorithm is $O(nmt_{look})$, where n is the number of actions matching B and m is the number of actions matching A. The fix algorithm is constant time.

The *all-matching* constraint set is quite useful for expressing requirements in which a particular action x does not *require* a corresponding activity in relation to it, but if corresponding actions do exist, they must have some particular temporal or causal relationship with respect to x. As an example, consider the following:

```
:constraint
 (all-matching-before
    :actions ((suspended-ceiling ?f) (finish-flooring ?f)))
```

In this case, if a suspended ceiling is built on a particular floor, it must be completed before the flooring is finished. However, if a different type of ceiling were built, no temporal relation would be enforced. As we will discuss in Section 7, such requirements are somewhat awkward to express in a STRIPS-based framework. Also note that *all-matching* constraints are often used in lieu of temporal/causal constraints. In particular, if other constraints are guaranteed to add all required actions of type A and B into a plan, an *all-matching* constraint can be used to order them rather than a temporal/causal constraint.

6.2.4 Decompose Constraint Form: decompose(A,Decomps)

The COLLAGE *decompose* constraint is analogous to task reduction in a traditional planner. Each *decompose* constraint provides an action descriptor, A, and a set of decomposition descriptors, *Decomps*. The constraint is satisfied if each action of type A is decomposed in exactly one of the possible ways provided by the decomposition descriptors. Each decomposition descriptor is composed of five parts:

- A set of action descriptors, *SubActions*. These describe the subactions into which an action of type A may be decomposed.
- A set of action descriptors, *FirstSubActions*, where *FirstSubActions* $\subset$ *SubActions*. These describe the "first" subactions of the decomposition.
- A set of action descriptors, *LastSubActions*, where *LastSubActions* $\subset$ *SubActions*. These describe the "last" subactions of the decomposition.
- A set of relations, *Relations*, between action descriptors in *SubActions*. These describe relations that must hold between the subactions in a decomposition.
- An optional condition, *Condition*, that constrains the situations in which a decomposition can be used. The form and use of *Condition* for a particular decomposition is similar to general constraint conditionalization, described in Section 6.3.

Truth Criterion: $(\forall\ a{:}A)\ \oplus_{\delta \epsilon Decomps}\ decomposed(a,\delta)$

where

$decomposed(a,\delta) \equiv Condition_\delta \wedge (\forall\ S \in SubActions_\delta)\ (\exists\ s{:}S)$

$[\ (subaction\ a\ s) \wedge$

$S \in FirstSubActions_\delta \supset (firstsubaction\ a\ s) \wedge$

$S \in LastSubActions_\delta \supset (lastsubaction\ a\ s)\ ] \wedge$

$(\forall\ (R,S1,S2) \in Relations_\delta)\ (\forall\ s1{:}S1)\ (\forall\ s2{:}S2)$

$[\ (subaction\ a\ s1) \wedge (subaction\ a\ s2)\ ] \supset (R\ s1\ s2)$

The following is also required:

Internal Coherence:

$(subaction\ a\ s) \supset$

$(firstsubaction\ a\ s) \vee (\exists\ first)\ [\ (firstsubaction\ a\ first) \wedge (first \Rightarrow s)\ ] \wedge$

$(lastsubaction\ a\ s) \vee (\exists\ last)\ [\ (lastsubaction\ a\ last) \wedge (s \Rightarrow last)\ ]$

External Coherence:

$[\ (b \Rightarrow a) \wedge (firstsubaction\ a\ first)\ ] \supset (b \Rightarrow first) \wedge$

$[\ (b \Rightarrow first) \wedge (firstsubaction\ a\ first)\ ] \supset (b \Rightarrow a) \wedge$

$[\ (a \Rightarrow c) \wedge (lastsubaction\ a\ last)\ ] \supset (last \Rightarrow c) \wedge$

$[\ (last \Rightarrow c) \wedge (lastsubaction\ a\ last)\ ] \supset (a \Rightarrow c)$

Internal coherence guarantees that the "first" and "last" subactions of a decomposition make sense – i.e., that all other subactions occur between some first and last subaction. If a decomposition is internally coherent, the first and last subactions of decomposition can used to relate all the subactions in the decomposition to other actions in the plan – i.e. as a framework for ensuring external coherence. In COLLAGE, internal coherence is required of each decompose constraint description and external coherence is enforced as part of the temporal closure mechanism.

Check Algorithm

Check(Plan)

For each action *a* possibly matching *A*
    If there is no relation *(subaction a s)* then
        Add *not-decomposed(a)* to *BugSet*
Return *BugSet*

Fix Algorithm

Fix(not-decomposed(a),Plan)

CHOICE POINT: Choose δ from *Decomps*
If $Condition_\delta$ does not hold, FAIL.
For all S in $SubActions_\delta$
    Create an action s that necessarily matches S
    Add *(subaction a s)* into *Plan'*
    If $S \in FirstSubActions_\delta$
        Add *(firstsubaction a s)* into *Plan'*
    If $S \in LastSubActions_\delta$
        Add *(lastsubaction a s)* into *Plan'*
For all *(R S1 S2)* in $Relations_\delta$
    For all *s1* such that *(subaction a s1)* and *s1* necessarily matches *S1*
        For all *s2* such that *(subaction a s2)* and *s2* necessarily matches *S2*
            Add *(R s1 s2)* into *Plan'*
Return *Plan'*

A slight variant of the *decompose* constraint form, *decompose-reuse*, is also in the COLLAGE library. This constraint will reuse existing actions as part of a decomposition instead of always creating new subactions. The fix algorithm is the same, except that we replace the step:

Create an action s that necessarily matches S

with:

PossibleSubactions $\leftarrow$ all actions that possibly match S
CHOICE POINT: Choose s from *PossibleSubactions* or create an action s matching S
Add binding relations into *Plan'* so that s necessarily matches S

The check method for a *decompose* or *decompose-reuse* constraint has complexity $O(t_{look})$. Excluding the cost of testing a decomposition condition, the fix cost for each particular decomposition is at worst $O(rn^2)$, where r is the number of relations and n is the number of subactions. In practice, $O(r+n)$ is more accurate.

Consider the following decomposition constraint from the office building domain, for finishing internal walls on a particular floor. In this case, there is only one valid decomposition and no conditionalization. The constraint description uses a Lisp notation for labeling action descriptors.

```
:constraint
  (decompose
    :action ((do-partitioning ?f))
    :decompositions
       ((:subactions (#1=(m-and-e-wall-services ?f)
                      #2=(drywall-studs ?f)
                      #3=(drywall ?f)
                      #4=(taping ?f)
                      #5=(painting ?f)
                      #6=(wall-fixtures ?f)
                      #7=(door-frames ?f)
                      #8=(doors ?f)
                      #9=(window-frames ?f)
                      #10=(glazing ?f))
         :first-subactions (#1# #2# #9#)
         :last-subactions (#6# #8# #10#)
         :relations ((#1# => #4#) (#2# => #3#)
                     (#2# => #7#) (#3# => #4#)
                     (#4# => #5#) (#5# => #6#)
                     (#7# => #8#) (#9# => #10#)))))
```

6.2.5 Pattern Constraint Form: pattern(PatternActions,Pattern)

The pattern constraint form is a unique descriptive mechanism that has no true analog in traditional planners. The check and fix algorithms for this constraint form are the most expensive and complex in the current COLLAGE library, with costs ranging from quadratic in the best case to exponential in the worst. Each pattern constraint provides a set of action descriptors, *PatternActions*, and a regular expression, *Pattern*, expressed in a language we define below. A pattern constraint is satisfied if all action instances that possibly match action descriptors in *PatternActions* are totally ordered and this total order satisfies (is a valid "parse" of) *Pattern*.

Truth Criterion: *parses(ActionSet,Pattern)*
where $ActionSet = \{ a \mid (\exists\ A \epsilon PatternActions)\ possibly\text{-}matches(a,A) \}$

To clarify this constraint form a bit, we begin with an example from the Blocks World (the current office building domain does not use this constraint form). An action of form `(pick ?x)` represents the act of picking up a block `?x`. An action of form `(put ?y ?z)` represents the act of putting a block `?y` onto a surface `?z`. The "rebind" expression provides information about which parameter descriptors can be rebound after each iteration around a "loop" in the regular expression.

```
:constraint
 (pattern
   :actions ((pick ?x) (put ?y ?z))
   :regexp  (((pick ?x) => (put ?x ?y))*=> [rebind ?x ?y])
```

This pattern gathers up all `pick` and `put` actions in a plan and requires that they alternate between `pick` and `put`, beginning with a `pick` action and ending with a `put` action. Moreover, each action that picks up a block must be immediately followed (in the sequence) by an action that puts that block down on some surface. For example, a valid string of `pick` and `put` actions is:

```
(pick a) => (put a b) => (pick c) => (put c a)
```

An invalid sequence is:

```
(pick a) => (put c a) => (pick c) => (put a b)
```

The regular expression language for pattern constraints is defined by the following syntax:

```
<expr> :== <term> | <term> <binary-rel> <expr>
<term> :== <factor> | <factor> <unary-rel>  <rebind>
<factor> :== ( <expr> ) | ( <expr> + <expr> ) | <action descriptor>
<binary-rel> :== => | ~> | >>
<unary-rel> :== *=> | *~> | *>>
<rebind> :== [rebind <var-list>]
```

The use of `*=>`, `*~>`, or `*>>` represents zero or more repetitions of a pattern fragment. The final action in each sequence matching a repetition must be related to the first action in the next matching sequence by the designated relation. The use of `+` indicates *disjunction* in the pattern – i.e. the pattern is composed of one of two prescribed subpatterns. Thus, a pattern of form

```
((pick ?x) + (put ?y ?z))*=>
```

would simply require all `pick` and `put` actions to be totally ordered. However, it would impose no particular ordering on them nor any required relationships between their parameters.

Activators: *PatternActions*

Initial Activation Setting: OFF

Check Algorithm

Check(Plan)

$\alpha \leftarrow$ All actions that possibly match any member of *PatternActions*
If *totally-ordered*(α) and *parses*(α,*Pattern*) then *BugSet* $\leftarrow$ { }
Else *BugSet* $\leftarrow$ *enforce-pattern*(α)
Return *BugSet*

Fix Algorithm

Fix(enforce-pattern(α),Plan)

Loop until α is empty
    NextPossibleActions $\leftarrow$ Set of actions that can occur first in the partial ordering formed by α in *Plan*
    NextPossibleActionDescriptors $\leftarrow$ Set of action descriptors that can follow in the current parse of *Pattern* that possibly match some action a in α.
    CHOICE POINT: Choose A from *NextPossibleActionDescriptors* and a corresponding a from *NextPossibleActions*
    Add bindings into *Plan'* so that a and A necessarily match
    Add the appropriate relation into *Plan'* between a's predecessor in the parse and a
    Remove a from α
If *Pattern* is not at a valid stopping point, FAIL
Return *Plan'*

This implementation of the pattern constraint has several important restrictions. First, it only adds relations between existing actions in a plan – it does not *create* actions in order to satisfy a pattern.[10] Second, this fix generates solutions in a lazy fashion. It finds and enforces one valid parse at a time and finds other solutions only if backtracking occurs within the internal fix search space. Thus, this algorithm provides no avenues for finding a "best" parse, except, perhaps, via the introduction of heuristics at the choice point.

The pattern check algorithm is actually a simplified version of the fix algorithm. After gathering α — an $O(nt_{look})$ operation, where n is the size of α — it checks to make sure that those actions are totally ordered (we use a simple $O(n^2)$ algorithm that assumes full

[10]Otherwise, the number of possible solutions could be infinite – e.g., for patterns with loops.

temporal closure) and can yield a valid parse. The parsing cost can range from $O(nm)$, where m is the disjunctive branching factor in the pattern parse, to $O(m^{n+1})$. In the fix algorithm, the cost is driven up by the operation of finding the next possible "first" actions in α. Each such operation costs at most $O(n^2)$, yielding a total cost ranging from $O(n^3m)$ to $O(n^2m^{n+1})$. In practice, we have found pattern constraint costs to be closer to polynomial rather than exponential.

6.3 Constraint Conditionalization and Binding Requirements

Two recent and powerful extensions to the COLLAGE constraint mechanism provide the ability to conditionalize constraint application and to impose additional binding requirements on the parameter descriptors used within a constraint. As discussed in Section 6.1.1, each execution of a check or fix is associated with a "current binding list" for a constraint's parameter descriptors. This binding list is passed through the execution of a check and fix and is refined as parameter descriptors are matched against parameter objects. Given *Condition* and *BindingReq*, the check and fix algorithms for a constraint are augmented in a way that refines usage of this list:

Augmented Check Algorithm

Augmented-Check(Condition,Check(Plan))

Test *Condition*, building a list *BL* of all combinations of bindings for
    parameter descriptors in *Condition*, so that *Condition* is satisfied.
For each *bl* in *BL*
    Set bindings to *bl*.
    Bugs ← *Check(Plan)*
    Add *Bugs* to *BugSet*
Return *BugSet*

Augmented Fix Algorithm

Augmented-Fix(BindingReq,Fix(Bug,Plan))

Plan' ← *Fix(Bug,Plan)*
Add bindings *BindingReq* into *Plan'*
Return *Plan'*

Normally, a check algorithm must consider all possible instantiations of a plan's parameter objects. In the augmented constraint check algorithm, *Check* will be applied within each of the binding contexts in which *Condition* is true. In essence, the augmented check algorithm

takes a constraint and breaks it into several sub-constraints, each pertaining to a specific binding context. It applies these sub-constraints individually, applying checks and fixes for each of them. The augmented fix algorithm, after successfully calling a particular *Fix*, will additionally impose all of the binding requirements in *BindingReq*.

COLLAGE requires *Condition* to be a list (interpreted as a conjunction) of boolean queries of the following forms:

`(action <action descriptor>)`	find a matching action
`(<relation> <action descriptor1>` `<action descriptor2>)`	find a matching relation
`(test <function>)`	tests a boolean function
`(fact <template>)`	queries the domain knowledge data base
`(make <parameter descriptor> <value>)`	creates a new parameter descriptor

Condition conjuncts are processed in linear order. As a result, the complexity of building *BL* is affected by the ordering of the conjuncts as well as the nature of each conjunct and its parameter descriptors. We require that all parameter descriptors supplied to boolean tests have constant-valued bindings. A simple example will demonstrate the use of these features.

```
:constraint
  (all-matching-before
     :condition ((action (lay-hall-flooring ?floor ?hall
                                            ?hallfloortype ?hallfloorcolor))
                 (test (> ?floor 0)))
     :actions ((lay-room-flooring ?floor ?room
                                  ?roomfloortype ?roomfloorcolor)
               (lay-hall-flooring ?floor ?hall
                                  ?hallfloortype ?hallfloorcolor))
     :binding ((color-match ?roomfloorcolor ?hallfloorcolor)))
```

If this *all-matching-before* constraint is activated and applied, the constraint condition will first find all `lay-hall-flooring` actions in the plan and creating a binding list consisting of all parameter descriptor binding combinations in which `?floor` is greater than 0. The `all-matching-before` check and fix will then be applied for each possible binding combination, additionally imposing the `color-match` binding requirement at the end of each fix. Even if the color parameters of two particular actions are not yet bound, the `color-match` binding requirement will be inserted into the plan, assuring that the colors match when they are finally selected.

Finally, the relationship between a constraint's *Condition* and *Activators* bears some further discussion. A constraint will only be activated if one of its activators are triggered. Only after the constraint has been selected for application will *Condition* be tested. If *Condition* is nonmonotonic, the following scenario is conceivable. First, a constraint C is activated and selected, but its *Condition* doesn't hold, so C is ignored. However, later, when *Condition* becomes true, C is not reactivated and is thus never appropriately satisfied.

In order to handle this type of situation, we have pursued the following solution: all action descriptors utilized in a *Condition* are added to the activator list of C. The rationale is that plan-related lookups are the most likely condition components to be non-monotonic. Indeed, currently, all data base facts and functions in *DomainKnowledge* are static.

6.4 Temporal Closure and Binding Propagation

All of the constraint algorithms in COLLAGE assume that each region plan is temporally closed and that binding requirements have been propagated. Of course, these operations have nontrivial cost. In this section, we consider the mechanisms utilized for these operations.

Our motivations for using full temporal closure were twofold. In our experience with GEMPLAN, we found that the system paid a large price for repeated temporal inferencing. For example, each time a relation of form `(a => b)` is added into a plan, the planner must ensure that `b` does not already precede `a`; otherwise, the result would be a temporally inconsistent plan. This "precedes" relation, being the transitive closure of `=>`, is thus both expensive and frequently called. When we designed COLLAGE, we decided to perform this closure explicitly and incrementally. Now all precedence tests can be made in near constant time.

The second motivation for maintaining closure was to test the efficacy of localization as a partitioning technique. Since plans are closed only on a regional basis, we conjectured that the price of full closure would be mitigated. We also anticipate that localization will also have a cost-reducing effect on binding propagation. Although we have yet not fully tested the strategic consequences of this approach, our current results are promising.

The mechanism used for incremental temporal closure is fairly straightforward. There are two types of closure: simple temporal closure and action decomposition closure (i.e. enforcement of *external coherence*). To achieve simple temporal closure, each time a relation `(a => b)` is added into a plan, COLLAGE finds all actions before `a` and all those that follow `b`. It then forms a cross product between these two sets and adds a new temporal relation (if it does not already exist) between each pair in the cross product. Action decomposition closure enforces the rules of external coherence defined in Section 6.2.4 and is performed in response to each assertion of a `firstsubaction` or `lastsubaction` relation, or any temporal relation imposed on decomposition-related actions. Both operations are at most $O(n^2)$, where n is the number of actions in a region plan. As a result, the total worst case cost is $O(n^4)$, since at most $O(n^2)$ relations can be added.

Each time a fix adds a new binding requirement into a region plan, COLLAGE propagates that requirement with respect to the binding network in that plan. COLLAGE's binding facility differs from typical CSP network implementations in several ways:

- The "overall" net is localized into a set of regional nets.
- Binding propagation is performed incrementally and allows for incremental addition of new variables into the region net.

- Only node and arc consistency are performed during plan construction, using a variant of the NC and AC algorithms described in [25]. Only at the end of the planning process, when final consistency is required, is path consistency performed.

Finally, notice that although COLLAGE maintains closure and propagates binding requirements incrementally within each $Plan_R$, these closure and propagation operations must also be performed when $Plan_R$ is integrated into the plans of R's ancestor regions. COLLAGE utilizes the same incremental closure and propagation algorithms for these inter-region consistency maintenance steps. The overall process is described in more detail in Section 8.

7 Action-Based vs. State-Based Planning

Our original interest in action-based representation and planning was rooted in the desire to develop natural ways of handling coordination requirements. We have found that people tend to think about activity coordination in terms of the actions being coordinated rather than in terms of the states surrounding those actions. One explanation for this may be the limited nature of human perceptive capabilities. If many activities are going on in parallel, it is often easier for us to observe individual actions and their temporal sequencing than it is to sense the highly dynamic, and in some cases, unpredictable and unobservable global domain state.

Consider the patterns of behavior expressible with pattern constraints. This kind of behavioral requirement is quite awkward to describe in terms of state, but is quite natural in terms of actions. Another class of action-oriented requirements are resource usage policies. For example, consider a "first-come-first-serve" policy. A plan may include "request" actions that register a request to use a resource and "serve" actions that utilize the resource. The "serve" actions must be ordered in the same order as their corresponding "request" actions. In order to encode this policy in terms of state, a request-queue state object must be utilized to represent the ordering of request actions. Formation of STRIPS-based action-descriptions that utilize and manipulate such a queue can be quite difficult to construct correctly [18]. In contrast, it is easy to represent this requirement directly in terms of action orderings:

```
:constraint
  (tempbefore
     :condition ((tempbefore (request ?x) (request ?y)))
     :actions ((serve ?x) (serve ?y)))
```

Another distinctive quality of action-based constraints is that they are compact and self-contained; requirements are stated within a single constraint. In contrast, STRIPS-based representations distribute the description of specific requirements among the preconditions and effects of many STRIPS-based action descriptions. Such distributed descriptions are often difficult to construct and maintain and are also more difficult to localize.

Besides being natural to use and compact, most of COLLAGE's action-based constraint forms are associated with cost-efficient plan construction algorithms. Moreover, we have

found that these forms are quite adequate for expressing the kinds of requirements we have encountered in real-world domains. For this reason, we have not incorporated an implementation of the modal-truth-criterion in COLLAGE.[11] The rest of this section attempts to demonstrate the adequacy of the action-based approach by showing how various aspects of state-based description can be encoded in an action-based framework. We use illustrations from the office-building construction domain, some of which contrast COLLAGE constraints with samplings from a SIPE specification for the same domain [13].

7.1 Goals

Instead of describing problem-specific goals in terms of desired goal states, COLLAGE utilizes action constraints. For instance, SIPE uses goals like *achieve(built-beam ?f ?c1 ?c2)* for each beam in the office building. In contrast, the constraint below is used by COLLAGE to generate high-level `build-beam` actions for each building "pod." Notice how the problem-specific aspects of a "goal" can be separated from the more domain generic aspects (every building must have beams – but each specific building has specific beam requirements) by using problem-specific information in the domain knowledge data base to conditionalize the application of action constraints. Thus, the constraint below can be used for all building problem instances. Each pod is a cube-like building-block; each office building is specified in terms of a set of pods. A pod is associated with a particular floor, has four corner points, and is composed of four columns, four walls, four beams connecting the tops of the columns, and a deck laid on top of the beams.

```
:constraint
 (action
   :condition ((fact (pod ?f ?c1 ?c2 ?c3 ?c4)))
   :actions ((build-beam ?f ?c1 ?c2)
             (build-beam ?f ?c3 ?c4)
             (build-beam ?f ?c1 ?c3)
             (build-beam ?f ?c2 ?c4)))
```

7.2 Preconditions

In general, temporal/causal, all-matching, and pattern constraints are used in lieu of preconditions. For example, consider a construction requirement that tile be laid before faucets are installed. In a traditional representation, this requirement would be encoded by requiring that a state condition *laid-tile* hold before each faucet-installation action. But since there may be only one ways of achieving *laid-tile* and no foreseeable way of undoing or removing it, it is easier to simply require that a temporal relationship hold between tile-laying and faucet-installation actions. For example, we could use a constraint of form

[11] In contrast, the GEMPLAN planner did include a traditional planning mechanism as one of its constraint forms – see Section 7.6.

```
:constraint
  (tempbefore
     :actions ((lay-tile) (install-faucet)))
```

Given this formulation, we could also decompose **lay-tile** in one of several ways, thereby achieving the same effect as allowing for many ways of achieving the state *laid-tile*. And if there *were* a tile-removing action, **strip-tile**, we could still use an action-based constraint to represent desired forms of behavior . For example, we might use a pattern constraint with the following regular expression:

```
( (lay-tile => strip-tile)*=> => lay-tile => install-faucet )
```

This would allow tile to be repeatedly laid and stripped, but ultimately, a **lay-tile** action must be the last tile-related action before **install-faucet**. In contrast, a state-based representation would describe how each of these action types "add" or "delete" *laid-tile*.

All-matching constraints have a interesting and expressive capability for describing certain kinds of precondition requirements. Consider the following constraint that requires embedded-slab utilities, if they are necessary, to be installed before the slab is poured:

```
:constraint
  (all-matching-before
     :actions ((install-embedded-slab-utilities ?s) (pour-slab ?s)))
```

In a SIPE implementation, this requirement must be expressed using *negative preconditions*:

- `(pour-slab ?s)` has the effect *(slab ?s)*
- `(install-embedded-slab-utilities ?s)` has the precondition *(not (slab ?s))*

The reason for this awkward use of negative preconditions is twofold:

> "(1) The latter activity does not actually require the former activity as an imperative prerequisite... For example, a slab can be poured regardless of whether embedded-slab utilities will be installed. This is different from other construction activities that have strong (hard) logical dependency, such as activities that follow one another based on the gravity support principle.
>
> (2) The former activity normally must precede the latter activity... For example, to install embedded-slab utilities after the slab has been installed would require extra tasks, e.g., destroying part of the slab..." [13]

7.3 Task Decomposition

Instead of using hierarchical task networks to describe goal decomposition, COLLAGE utilizes the decompose constraint to decompose high-level actions. Because decompositions can be conditionalized, this constraint form can be quite powerful. The analogue in SIPE is the use of operator plots.

One advantage of the decompose constraint over plots is that it retains high-level actions in the plan rather than replacing them. As a result, constraints can be imposed on activities at mixed levels of detail; atomic and nonatomic actions can be interrelated without restriction. One difficulty encountered by the SIPE implementation effort for the office building domain were interactions between "replacement-based" task decomposition and required limitations on the use of parallel links. SIPE-2 was a direct outgrowth of modifications required to deal with these difficulties. The following are excerpts from a study that analyzed these interactions [13].

> "SIPE did not produce the least-constrained, correct plan... One of the basic restrictions was that given a set of parallel subplans, SIPE will only reorder them by putting a whole subplan before or after the others. This restriction greatly reduced the number of possible orderings, but it could not produce all possible permutations of the actions in the subplans... it proved too restrictive in real-life construction problems...
>
> ...The representation of parallel links is complicated by the use of hierarchical abstraction levels. There may be a number of parallel links... at any one node and copying these links down to more detailed levels raises a problem since some nodes may be expanded to a more detailed level while other nodes may not be...
>
> ...Unfortunately, the new extension to SIPE to produce the most parallel plan uncovered a new problem involving the introduction of redundant actions in parallel subplans. Duplicating actions in parallel subplans intensifies significantly when an action needs to be linked to several actions in parallel, such as the Office Building project. Wilkins... modified his planner to overcome this problem."

Interestingly, rather than using preconditions, SIPE also used plots to describe temporal requirements. This usage is similar in spirit to the use of action-based constraints to describe preconditions. For example, consider the following operator for `DO-BEAM`:[12]

[12]Note that this domain encoding utilizes 3-dimensional coordinates to describe building locations, rather than using a floor level with a two-dimensional coordinate.

```
OPERATOR: do-beam
ARGUMENTS: beam1, column1, column2,
           location1, location2, location3, location4;
PURPOSE: (done beam1)
PRECONDITION: (beam-location beam1 location1 location2),
              (column-location column1 location3 location1),
              (column-location column2 location 4 location2);
PLOT:
   PARALLEL
      BRANCH 1:  GOAL: (done column1);
      BRANCH 2:  GOAL: (done column2);
   END PARALLEL
   PROCESS:
      ACTION: build-beam;
      ARGUMENTS: beam1;
   EFFECTS: (done beam1);
END PLOT
END OPERATOR
```

Of course, building columns is not truly a logical decomposition of building a beam – it is really a precondition or temporal requirement. In COLLAGE, this requirement is explicitly represented by the following temporal constraints:

```
:constraint
 (tempbefore
   :actions ((build-column ?f ?c1) (build-beam ?f ?c1 ?c2)))
:constraint
 (tempbefore
   :actions ((build-column ?f ?c2) (build-beam ?f ?c1 ?c2)))
```

7.4 Filter Conditions

Depending on their underlying semantics, the use of "filter" conditions can be mimicked in COLLAGE using constraint conditionalization. For example, suppose we have a filter condition *have-paint* for each `paint-wall` action in a plan. Although we do not wish to add paint-acquisition actions into the plan, the filter condition ensures that there is paint available to successfully perform `paint-wall`. In COLLAGE, we could get the same effect by conditionalizing the constraint that generates a `paint-wall` action. For example:

```
:constraint
 (action
   :condition ((test (have-paint)))
   :actions ((paint-wall ?f ?c1 ?c2)))
```

7.5 Resources

Another important type of domain requirement — one that is often associated with state-based description – deals with resource usage. In particular, states are usually used to represent the status of a resource at each point in time. In COLLAGE, resources are currently handled in two ways:[13] (1) via binding requirements; and (2) by treating regions as resource "objects" that constrain their own behavior.

If an action is associated with parameters that represent its required resources, binding requirements on those parameters can be used to control resource choices. Similarly, if a parameter is used to represent the metric time point at which an action occurs, binding requirements can be used to constrain the times at which actions occur.

The use of regions as resource objects results in a rather interesting form of "object-oriented" planning. Each "resource" region is associated with constraints (such as the "first-come-first-served" constraint described earlier) that limits its action behavior and thus the use of its associated resource. For example, consider the region structure discussed in Section 4 and depicted in Figure 2. We provide a fragment of that description below.

```
(defregion (all-columns all-columns-type)
  :subregion (:generate (column-builder column-builder-type
                         :limit max-number-of-column-builders)))

(def-region-type column-builder-type
 :action-type (build-column floor coord))
```

We can think of the `column-builder` regions as resources – i.e. the contractors who actually build columns. If desired, we could associate each `column-builder` region with additional constraints that limit individual `column-builder` activity. Since the `all-columns` region generates new `column-builder` regions, it too could be associated with constraints (or, possibly, region-generation heuristics) that control the allocation of column builders to specific tasks.

7.6 Representing STRIPS-Based Constraints

While we advocate using action-based formulation whenever possible, if state-based representation is truly necessary to describe certain domain requirements, several options can be pursued in a COLLAGE-like framework. One is to directly incorporate a "STRIPS" constraint form in the constraint library. This path was followed in the GEMPLAN planner [19] and its implementation is described below. Another path would be to use a set of conditionalized action-based constraints to effectively enumerate some of the possible "fixes" for the modal truth criterion. For example, suppose that we have a condition p with "adder" `A` and

[13] In the future, we plan to extend COLLAGE to better handle more complex resource requirements – e.g. those for handling continuous resources. Some form of "resource state" profile may have to be integrated.

"deleter" D. If we wish p to be a precondition to action E, we could use constraints of the following form (although this example is fairly simplistic, it illustrates the point):[14]

```
;; Simple establishment
:constraint
 (cause
  :condition ((not (fact (initial-state p))))
  :actions ((A) (E))))

;; Promotion, Demotion
:constraint
 (pattern
  :condition ((action (D))
              (cause (A) (E))
              (not (before (A) (D))) (not (before (D) (A)))
              (not (before (E) (D))) (not (before (D) (E))))
  :actions ((A) (D) (E))
  :regexp ((D => A ~> E) + (A ~> E => D)))
```

In contrast to the above action-based approach, a "STRIPS" constraint form was directly integrated into the GEMPLAN constraint library. Each such constraint requires a state-based condition to be true at some specified point in the plan – either at the end of the plan (a goal) or prior to specific actions (preconditions). In order to represent state conditions, GEMPLAN associates each region type with *predicate definitions*, in addition to action type descriptions and constraints. Each predicate definition includes a list of conditionalized adder and deleter descriptors. For example, in the GEMPLAN implementation of the blocks world, the following definition of *clear(B)* is used:

```
(predicate-definition
  :predicate (clear ?B)
  :adders (((pick ?Y) (on ?Y ?B))
           ((put ?B ?Z) true))
  :deleters (((put ?W ?B) true)
             ((pick ?B) true)))
```

Each adder or deleter descriptor provides: (1) an action descriptor which can add or delete a literal of the specified type; and (2) the condition under which such an action adds or deletes the literal. For example, an action of form `(pick ?Y)` adds `(clear ?B)` if `(on ?Y ?B)` is necessarily true just before it occurs. An action of form `(put ?B ?Z)` always adds `(clear ?B)`, and actions of form `(put ?W ?B)` or `(pick ?B)` always delete `(clear ?B)`.

Notice how this formulation makes conditional effects quite easy to describe. It is easy, for example, to state that an action adds a particular literal P in some contexts and another

[14]Note the use of negation-as-failure in the condition. Translating STRIPS-based requirements in this way would probably require some extensions to the constraint library and the form of constraint conditions.

literal Q in others. In addition, defining a predicate in one place (instead of "distributing" its definition among the preconditions and effects of several STRIPS-based action descriptions) makes it easier to localize. For instance, alternate definitions of "clear" might be used in different regions.

Given a predicate definition for a precondition or goal, the check method for a STRIPS constraint is essentially a test of the modal truth criterion. The fix methods correspond to the plan construction operations in traditional planners: promotion, demotion, separation, and establishment via the use of existing actions or via the addition of new actions.

8 Localized Search

COLLAGE controls the application of constraint checks and fixes by searching through a set of search spaces, one for each region. Each search node in a region space is associated with a region plan – i.e., the plan constructed thus far for that region. A node represents a branching point in the plan-construction process that deals with one of the following kinds of choices:

- *Constraint choice:* given a set of activated constraints, which constraint to apply next.
- *Bug choice:* given a set of constraint bugs, which bug to tackle next.
- *Fix choice:* given a constraint and bug, which fix method to apply.
- *Internal fix choice:* a choice point within a fix.

At a global level, COLLAGE must also decide which region's space it should be searching at any given point in time. As a result, COLLAGE must also search a global space, where each node is associated with the choice of which region to search. Figure 6 depicts these two search spaces or "levels." Together they form one large search space; in essence, the global space is composed of region space fragments. Each of these fragments is called a *region search incarnation.*

Notice that each region R may have several incarnations within the global space. Each of these search fragments "reincarnates" or continues search within R's space, further applying R's constraints in the pursuit of constructing $Plan_R$. Global search flows between region incarnations depending on how fixes activate region constraints. Recall that a fix applied within region R may activate constraints both within R and other regions in the domain. All of these regions may then become candidates for further search.

In this section we fully describe the COLLAGE localized search mechanism: how it controls search at both the global and region levels, and how it maintains overall plan consistency. The overall goals of localized search are twofold:

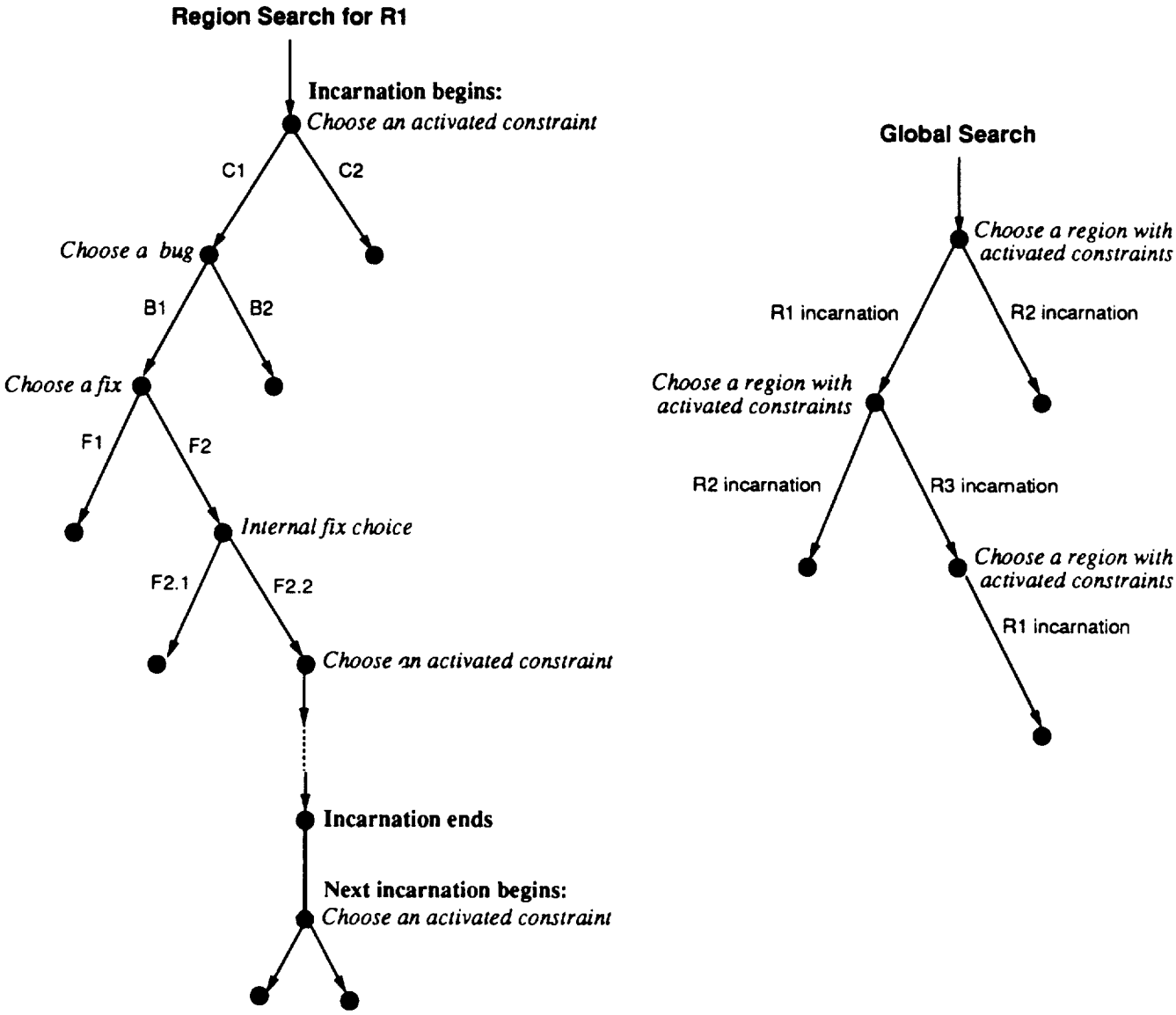


Figure 6: Search Frameworks

- **Correctness** – *make sure that when planning is done, all region constraints are satisfied by their region plan.*
- **Consistency** – *make sure that the "global" plan, consisting of all region plans, is consistent.*

Correctness is assured in COLLAGE by maintaining a valid account of constraint activation and making sure that all activated constraints are addressed. This task is performed via the use of *region agendas.* Consistency is assured in COLLAGE by making sure that region plan modifications are propagated and appropriately integrated into other, related, plans. This task is performed via the use of a *consistency agenda.*

8.1 Region Search

We begin by describing search within a region incarnation, as depicted in Figure 7. The first step performed in any incarnation for a region R is the *consistency update* step. This operation incorporates pertinent plan information from other regions into R's plan and also performs necessary inter-region temporal closure and binding propagation operations (see Section 8.2). After forming a consistent plan, search within R repeats a cycle of the form "choose an activated constraint; choose a constraint bug; apply a fix." This cycle continues until all activated constraints and their bugs are satisfied or until a search heuristic terminates search for that incarnation (say, in order to pursue planning in another region).

In order to keep track of activated constraints, pending bugs for each constraint, constraint fixes that have been tried, and internal fix choices that have been tried, each incarnation is associated with a *region agenda.* The incarnation search algorithm maintains and updates this agenda to ensure that all possible constraint/bug/fix options are explored as necessary. In practice, this agenda is stored in a distributed fashion; constraint-activation information is stored within a region plan and information about outstanding bugs, fixes, and internal fix choices is associated with R's search nodes.

The constraint activation/deactivation mechanism, in particular, bears some further clarification. Constraint activation information for a constraint C is stored in those local plans that "activate" C; *each local plan will contain an "activation" for each constraint activated by the information associated directly with that local plan.* Thus, $LocalPlan_R$ can be associated with constraint activations for any constraint in R as well as any of R's ancestors. The underlying constraint activation mechanism in COLLAGE must utilize global information about region structure and constraints to ensure that all appropriate constraints are activated. Constraint *deactivations* are handled in much the same way. A *deactivation* for a constraint C of region R is stored in $LocalPlan_R$ when that constraint has been satisfied. Given this localized storage of constraint activation and deactivation information, a constraint C is considered to be active relative to a particular plan depending on the activations and deactivations inherited by that plan.

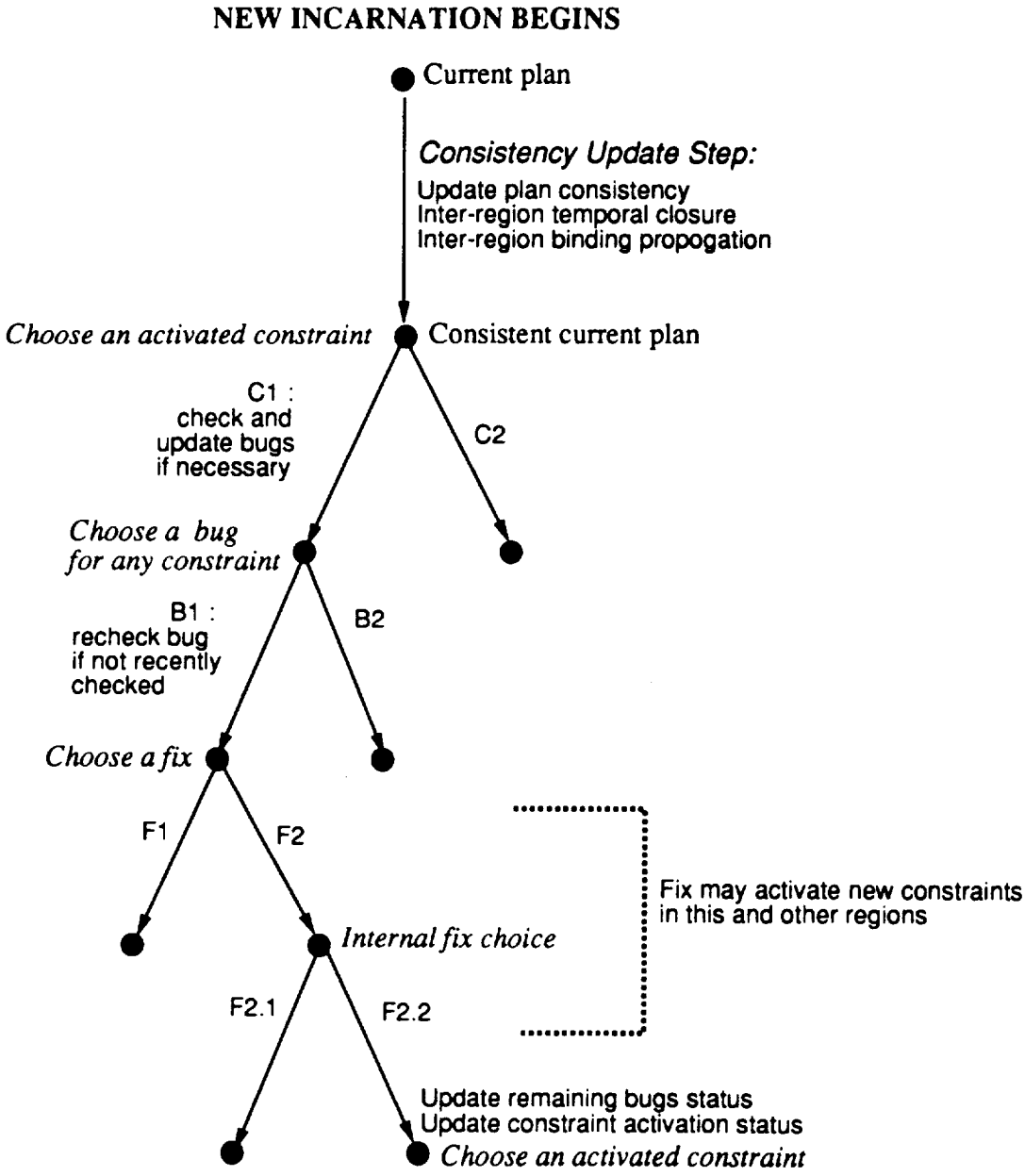


Figure 7: Search in a region incarnation

```
Search-Incarnation(R)

CurrentPlan ← Incorporate-Consistency-Updates(R)
While there are active constraints in CurrentPlan and
no heuristic requires termination of this incarnation
    Constraints ← Active constraints in CurrentPlan
    CHOICE POINT: Choose C from Constraints.
    If C has not been checked since its last activation then
        Bugs ← Check_C (CurrentPlan)
        Add Bugs to region agenda
    AllBugs ← All bugs in region agenda
    If AllBugs is not empty then
        CHOICE POINT: Choose B from AllBugs
        Fixes ← All possible fixes for B
        CHOICE POINT: Choose Fix from Fixes
        CurrentPlan ← Fix(B,CurrentPlan)
        Remove B from bugs in region agenda
        If all bugs for a constraint C have been removed and
            there are no activations since it was last checked,
            Deactivate C in LocalPlan_R
Return CurrentPlan
```

Notice that backtracking through the choice points in a region incarnation will occur when a fix fails. If all possible combinations of constraint orderings, bug orderings, and fix orderings within an incarnation fail, the search incarnation itself will fail. This will then cause backtracking into the global space, and ultimately, through previous incarnations in that space.

Notice that the above algorithm allows outstanding bugs for all constraints to be tackled in any order. That is, bug fixes for different constraints can be interleaved. The default search heuristics in COLLAGE, however, do not allow this level of flexibility. Once COLLAGE chooses a constraint and checks it, incarnation search will attempt to satisfy all bugs returned by that check before it considers bugs for any other constraint.

8.2 Global Search

As we have described, search within a region incarnation is concerned with satisfying constraint bugs. In contrast, the global search space tries to assure that all regions with active constraints are eventually incarnated and that all reasoning within an incarnation is performed on a consistent plan. The basic scheme for maintaining consistency is fairly simple:

Before searching a region incarnation, make sure the region's *current plan* is consistent with respect to the current plans of all other regions. After completing a region incarnation, take note of which local plans have been changed. Update the *consistency agenda* to ensure that these local plans are ultimately incorporated into other relevant regions.

The simplicity of this scheme is based on the fact that region incarnations are searched in sequence.[15] The notion of a *current plan* is also integral to this approach. The current plan of a region R is always *the plan associated with the last node visited in an R incarnation* – i.e., its most recent plan. Thus, if search backtracks out of an R incarnation, R's current plan must be reset to its previous value – the outgoing plan of its previous incarnation, or the initial R plan (usually empty), if no such incarnation exists.

The structure used to keep track of outstanding consistency-related information is the *consistency agenda*. It consists of a set of pairs of form `(R1 R2)`, which indicate that the current plan of `R2` must be integrated into the current plan of `R1`. Note that the consistency agenda is truly a global data structure. As a consequence, it must be updated every time backtracking occurs in the global space (which is, in practice, rare). In contrast, the region agenda is stored in a node-relative and/or plan-relative fashion, enabling backtracking to occur without any explicit region-agenda modification.

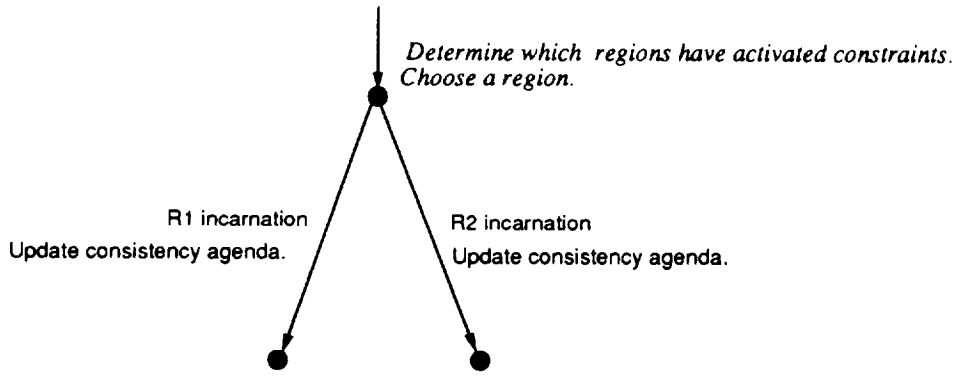


Figure 8: Global Search

The following algorithm describes the top level global search algorithm, as depicted in Figure 8. *Global-Search* assumes that, at the start of planning, the initial plan for each region R is associated with the initial activation status for each of R's constraints. At each node in the global space, the set of regions with active constraints is determined as follows. Using the current plans for each region and information in *ConsistencyAgenda*, the most recent

[15]While COLLAGE currently searches the various regional spaces sequentially, shifting from one region space to another, a localized search framework could also potentially serve as a natural testbed for distributed reasoning.

local plans for each region are found. From these, the most recent constraint activations can be determined.

Global-Search

While there are regions with activated constraints
    CHOICE POINT: Choose a region with active constraints, R
    CurrentPlan ← Current plan for R
    NewCurrentPlan ← *Search-Incarnation(R)*
    Update-Consistency-Agenda(R,CurrentPlan,NewCurrentPlan)
Perform any remaining consistency updates
Return final global plan, consisting of current plans for all regions.

Below are the algorithms used for updating the consistency agenda and for creating a newly consistent plan. Note that, in contrast to the more typical consistency pair `(R S)` (where `S` is a descendant of `R`), pairs of form `(S R)` are also used. These ensure that `R`'s changes to `S`'s local plan are made aware to `S` itself. *Update-Consistency-Agenda* is also used when search backtracks within the global search space, in order to notify all relevant regions that a region incarnation plan has been "backed out." Finally, note that COLLAGE utilizes a more efficient version of *Incorporate-Consistency-Updates* than the one presented below. This alternative algorithm orders consistency updates so that they occur in a bottom up fashion.

Update-Consistency-Agenda(R,OldPlan,NewPlan)

ChangedRegions ← All regions that have a different local plan
                    in *NewPlan* than in *OldPlan*.
For each S in *ChangedRegions*
    Add `(S R)` to *ConsistencyAgenda*
    Find all regions T such that *descendant(T,S)*
    For each T
        Add `(T S)` to *ConsistencyAgenda*

```
Incorporate-Consistency-Updates(R)

CurrentPlan ← Consistency-Update(R)
CurrentPlan ← Temporally-Close(CurrentPlan)
CurrentPlan ← Propagate-Bindings(CurrentPlan)
Remove all pairs (R S) from ConsistencyAgenda
Return CurrentPlan
```

```
Consistency-Update(R)

CurrentPlan ← Current plan for R
For each pair (R S) in ConsistencyAgenda
    UpdatedPlans ← Consistency-Update(S)
    Integrate UpdatedPlans into CurrentPlan
Return CurrentPlan
```

8.3 Benefits and Pitfalls of Localized Reasoning

Consider a search space in which each node is associated with a plan and each arc is associated with a plan-construction operation. There are at least three ways of improving planning costs in this framework:

1. *Reducing search space size*, by lowering the branching factor at each node.
2. *Lowering the cost of each plan-construction operation.*
3. *Search heuristics* that guide the order in which plan-construction operations are applied. This kind of improvement can reduce backtracking within the space and may also improve solution quality.

Localization can be viewed from several perspectives. As a technique for domain representation, it provides a means for partitioning the overall domain representation, thereby defining the scope of domain constraints. In the plan representation dimension, localization provides guidelines for partitioning a plan into plan fragments. The plan-construction algorithms may then be applied to these smaller fragments, rather than the entire global plan. Finally, in the control dimension, localization may be viewed as a heuristic for partitioning the overall search space, for reducing the branching factor at each search node, and for guiding how that space is searched. Thus, in terms of the potential search benefits described above, localization can achieve all three:

1. Since only region constraints can be applied within each region incarnation, localization reduces search space size by *limiting the branching factor at each node.*

2. Constraint fixes (i.e. the plan-construction operations) are applied to smaller region plans, thereby *reducing plan-construction cost.*

3. By using region agendas that monitor constraint activation, only those constraints relevant to changed portions of the plan are applied at each point in the reasoning process. Moreover, search shifts between region incarnations depending on the activation status of region constraints. Thus, localization serves as a *search heuristic* that controls the order in which plan-construction operations are applied.

In order to better understand the benefits and tradeoffs of localization, we have studied the technique both analytically and empirically. In [21], a detailed complexity analysis is provided, as well as some empirical results for GEMPLAN. Below, we summarize the results of these studies and provide some new empirical results for COLLAGE. As we will discuss in Section 9, we are also embarking on a much deeper empirical study of localization, using an enhanced version of the office building domain as our testbed.

Since the cost of localized search is very dependent on the constraints and structure of a particular domain, the "general" complexity analysis described in [21] had to be performed within a somewhat idealized domain scenario. The search cost of a non-localized domain was compared with that of the same domain, partitioned into a set of m regions, $R1...Rm$, each of which overlaps by some factor k with another region G (see Figure 9). In particular, k is the number of actions in the final plan within each region of overlap. There are a total of n_c constraints, and they are partitioned and distributed among G and $R1...Rm$. The number of regions with constraints, $m+1$, provides a measure of the amount of localization in this domain. G's final plan size, mk, provides a measure of regional overlap or sharing.

Complexity results were calculated for best-case and worst-case search, assuming that all constraint algorithms are either constant, linear, quadratic, or exponential in cost relative to plan size. Best-case is the cost of one path through the search space (no backtracking), and worst-case is the cost of the entire space. Table 1 provides the results of this analysis. The term s is the size of the final plan. The term n_f is the number of available fixes for each constraint. The number of incarnations was assumed to be $\frac{mk(m+1)}{n_c}$ for G and $\frac{s(m+1)}{mn_c}$ for $R1...Rm$. Finally, C is the cost of maintaining consistency, which is assumed to be $O(m^2k)$.

These results showed that, for this highly idealized scenario, localized search is in most cases much better than non-localized search. The only exceptions are constant-complexity best-case search (when there is no reduction in the amount of the space searched *nor* in plan-construction cost) or when the cost of consistency maintenance overshadows other costs. The amount by which localized search wins over non-localized search is proportional to m (the degree of localization), but inversely proportional to mk (the amount of overlap). Thus, increased localization is always worthwhile, *except for resulting increased consistency*

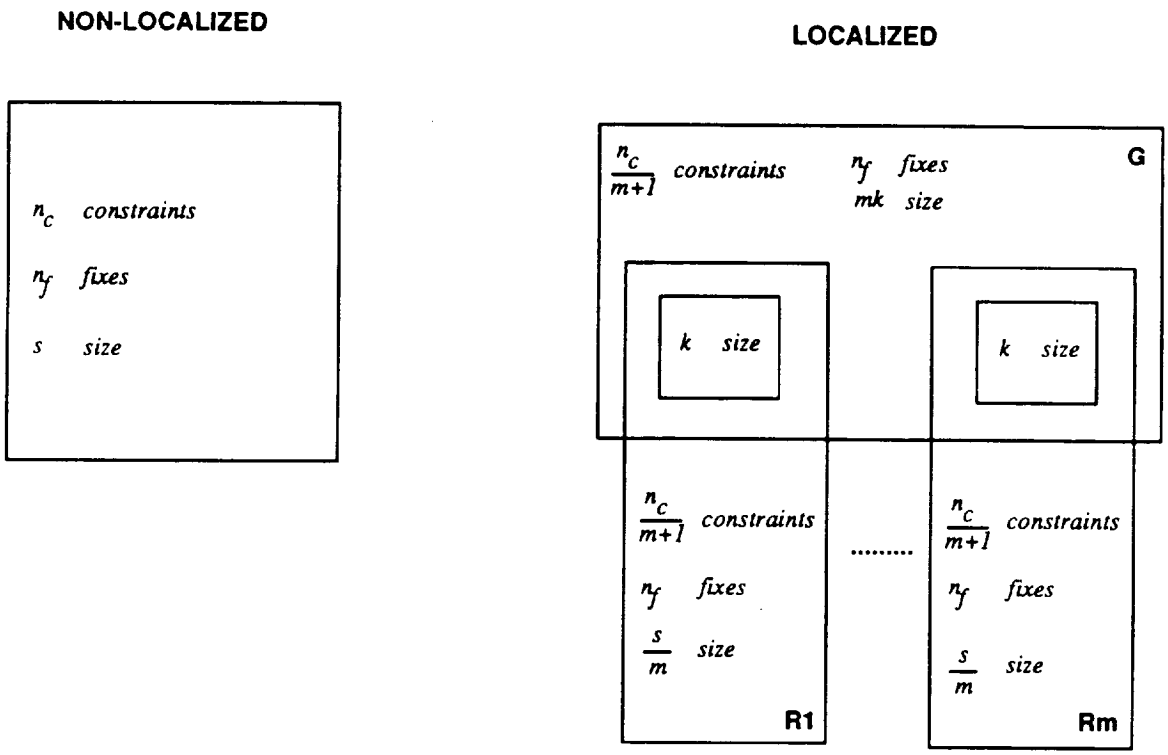


Figure 9: Non-Localized and Localized Domains

Constraint cost for plan of size i	*Non-Localized (best-case)*	*Localized (best-case)*	*Non-Localized (worst-case)*	*Localized (worst-case)*
constant (b)	$2bs$	$2b(s+mk)+C$	$b(n_c n_f)^s$	$b(m(\frac{n_c n_f}{m+1})^{\frac{s}{m}} + (\frac{n_c n_f}{m+1})^{mk}) + C$
linear (bi)	bs^2	$b(\frac{s^2}{m} + (mk)^2) + C$	$bs(n_c n_f)^s$	$b(s(\frac{n_c n_f}{m+1})^{\frac{s}{m}} + (\frac{n_c n_f}{m+1})^{mk}) + C$
quadratic (i^2)	$\frac{2}{3}s^3$	$\frac{2}{3}(\frac{s^3}{m^2} + (mk)^3) + C$	$s^2(n_c n_f)^s$	$\frac{s^2}{m}(\frac{n_c n_f}{m+1})^{\frac{s}{m}} + (mk)^2(\frac{n_c n_f}{m+1})^{mk} + C$
exponential (b^i)	$2b^s$	$2(mb^{\frac{s}{m}} + b^{mk}) + C$	$(bn_c n_f)^s$	$m(\frac{bn_c n_f}{m+1})^{\frac{s}{m}} + (\frac{bn_c n_f}{m+1})^{mk} + C$

Table 1: Complexity Results

maintenance costs. The gains of localized search become exponential as the complexity of the constraint algorithms increases and the amount of the space actually searched increases.

Empirical tests have been carried out with both GEMPLAN and COLLAGE that support these analytical results. The graph in Figure 10 depicts results from a COLLAGE test suite, using three different localizations for the office building domain: a non-localized partitioning, a localized partitioning, and a partitioning with a more moderate level of localization. The graph provides results for an office building ranging in size from one to eleven floors, with an identical pod structure (floor plan) on each floor.

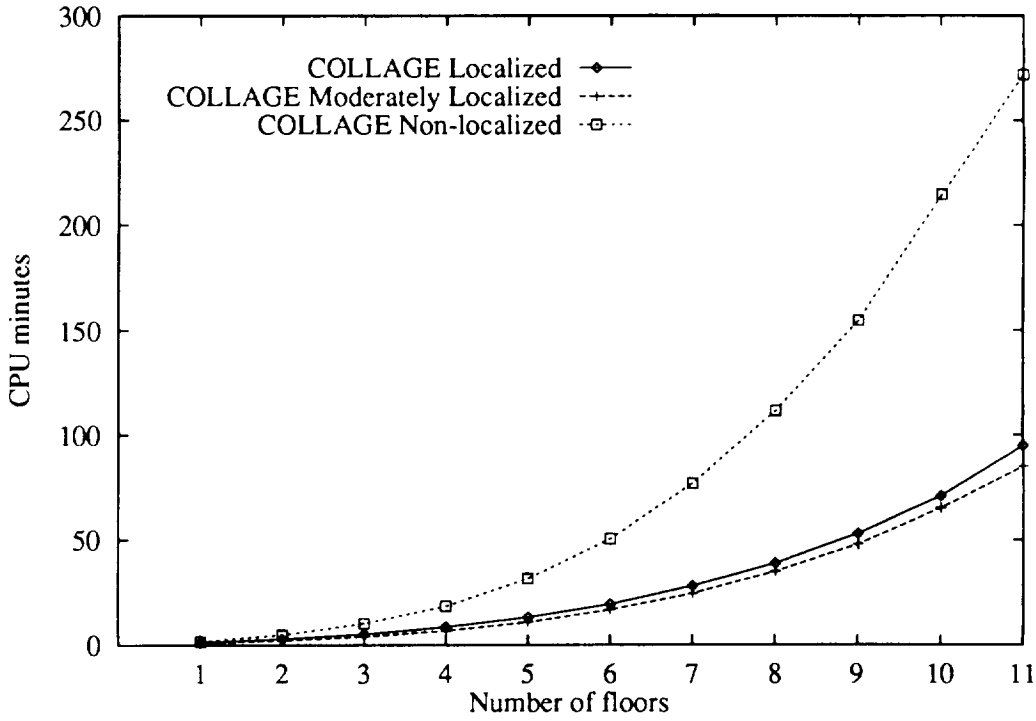


Figure 10: Office Building Test Results

Since there is no backtracking in this domain, most of the savings attained by the localized versions of the domain are attributable to a reduction in plan-construction cost and good search heuristics. Even though the plan-construction costs are fairly low in this domain (linear or polynomial for all constraints), the results show that localization can provide impressive benefits, except for the added expense of consistency maintenance in more highly partitioned domains. In a previous study with GEMPLAN on a house construction domain [21], empirical results also showed that the cost of consistency maintenance becomes less important as plan size and search space size increases.

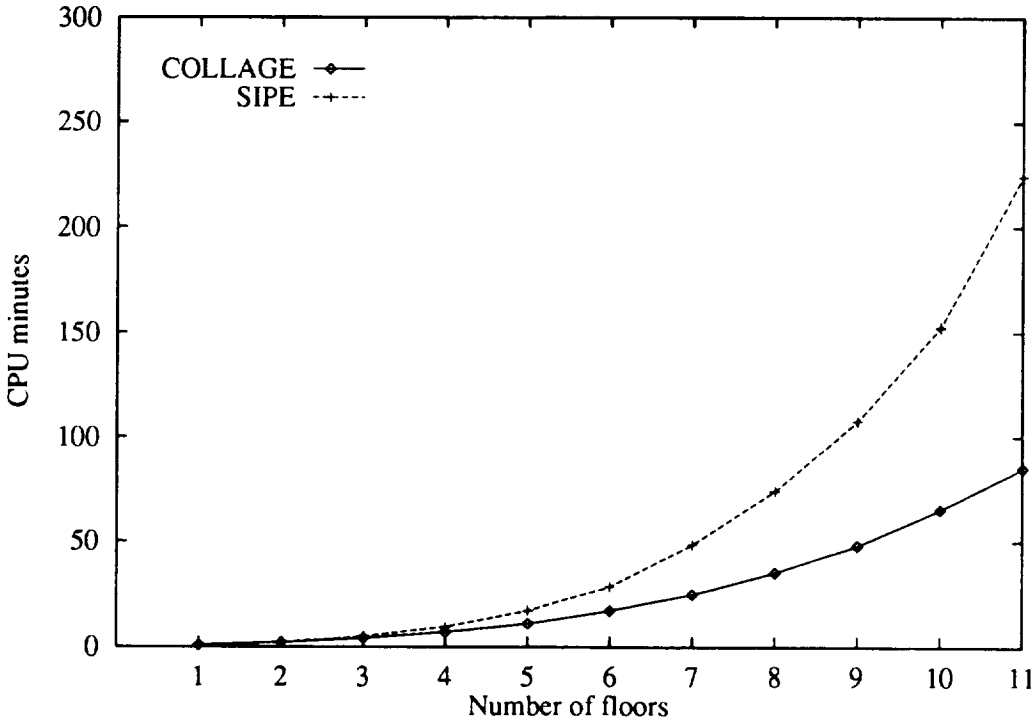


Figure 11: COLLAGE and SIPE test results

Finally, Figure 11 depicts planning costs for SIPE and COLLAGE on the same test suite of office buildings (using the best COLLAGE localization). The run times for SIPE are taken from [13]. Although these results are admittedly for an entirely different planner implemented on different hardware, it is clear that the derivative of the COLLAGE curve increases much more slowly than that of the SIPE curve. Most of this result is attributable to the use of localized search; another factor may be the relative efficiency of the COLLAGE plan construction algorithms.

8.4 Localization vs. Abstraction

Both localization and state-abstraction [3, 14, 16] may be viewed as heuristics for reducing planning search cost. Abstraction techniques explicitly break the problem-solving process up into "abstraction levels." The planning process starts by creating a plan at the highest level of abstraction. This high-level plan then serves as a starting point for planning at the next level of detail, with this process continuing until a plan at the lowest level of detail is formed. At each level, more information is added into the problem definition ("visible" state conditions and actions) to create a more complex planning problem. Since the use of abstraction levels inherently controls the order in which pieces of the problem are tackled, it can be viewed as a search heuristic. In earlier stages of the reasoning process, only "higher" level plan-construction operations, which involve high-level actions or conditions, are applied. This set is expanded as the problem and domain definition is expanded.

Although abstraction limits the set of applicable plan-construction operations at higher-level stages of the search process, a reduction of the number of applicable plan-construction operations (and a reduction of the application scope of these operations) is not *guaranteed* once a domain is fully expanded. As a result, planning at the lowest level of detail is still "global." To cope with this problem, abstraction-derivation techniques have been developed to guarantee properties such as *monotonicity* [15]. A monotonic hierarchy is guaranteed to be free from abstraction-level interactions, thereby guaranteeing a reduction in the search-space branching factor at each level, and potentially, the scope of each plan-construction operation as well.

In contrast to abstraction, localization divides a planning problem according to the inherent scope of its constraints. In practice, a localization structure is influenced by a broad set of criteria, not just "abstractness." Thus, localization can be viewed as a way of creating and simultaneously utilizing several different "kinds" of abstraction levels.

Most importantly, however, localization allows for domain regions that overlap and interact. Researchers using abstraction hierarchies have found that, in realistic domains, it is difficult to attain a monotonic abstraction-level partitioning. Because real-world domains almost always manifest natural forms of interaction, the quest for monotonicity is thwarted by a resulting collapse in the abstraction hierarchy. In contrast, localization embraces the notion that real-world domains cannot be neatly decomposed into independent regions. The localized search technique explicitly provides methods for coping with this regional interac-

tion, shifting search from region to region, rather than requiring that the planning process be neatly partitioned and monotonic. Thus, despite region interaction, all three kinds of search benefits can be attained in a localized search framework.

9 Current Work

Our current research with COLLAGE is progressing on several fronts. In addition to extending the constraint library in order to handle our two application domains, we are also enhancing the COLLAGE architecture in several ways. This section discusses three of the current foci of the COLLAGE project: "flexi-time" constraint activation and satisfaction, user integration into the planning process, and continued study of localized search.

9.1 Flexi-Time Constraints

Because of the coordination-intensive nature of our target domains, it is important to do most planning in advance of execution. For instance, the general contractor at a building site must plan most of the construction process in advance – a large structure cannot be built "reactively." However, complex, real-world domains also require run-time plan modification. This kind of reasoning can take at least two forms: (1) Some constraints cannot usefully be applied until run-time. Such constraints should be deferred until they are truly applicable or satisfiable. A typical example is a run-time dispatch constraint that controls access to resources. (2) Unanticipated situations resulting from run-time errors, user intervention, an incomplete domain theory, or environmental factors may trigger the need to make plan repairs.

In order to meet these requirements, we are currently working on extending the COLLAGE architecture to blend pre-execution search-based planning with more dynamic forms of reasoning.[16] We term this fusion of pre-planning with run-time reasoning *flexi-time constraint satisfaction*. The intuition is that a constraint should, in principle, be applicable at any time relative to execution. During pre-planning, constraints are triggered primarily by plan modifications made by the planner. However, constraints could also be triggered and applied in response to the run-time environment or the user. For example, the environment may cause unexpected modifications to the plan, or the user may decide to alter the domain and problem specification. The incorporation of flexi-time constraint satisfaction will also enable COLLAGE to emulate reactive planning architectures. For example, given decompose constraints that are reactively applied, along with an enhanced dynamic knowledge base, COLLAGE could emulate systems such as PRS or RAPS.

[16]Since our target domain class is not highly dynamic, we are not focusing on issues such as time pressures in run-time reasoning. Instead, our goal is to maintain plan correctness while providing flexibility in plan construction and repair.

However, backtracking in a plan-space search framework (where the order in which actions are added into the plan has no relationship with the order in which they are executed) becomes problematic once plan-execution has begun. How can one backtrack over a node if portions of the plan associated with that node have already been executed? We believe that the best solution is to conduct run-time planning much the same way a human would. Once execution has begun, a plan should be "patched." Information gleaned from a record of the prior search space may be useful, but backtracking into the prior search space is not. A similar tactic is taken in recent work on plan reuse and modification [12].

Our current strategy for flexi-time reasoning in COLLAGE is to create a "reformed" search framework each time plan modifications are made at run-time. Each "reformation" may be viewed as a new, global search framework. It begins with an initial node that encompasses new user directives (in the form of modifications to the plan or domain description) and other externally-motivated plan modifications.

Once reformed, constraint-satisfaction search may proceed much as during pre-planning search, tackling constraints that may have been activated in response to reformation changes. Other constraints that must be tackled are those that may have become violated due to the reformation process, even though they were not explicitly activated by it. Towards this end, we will be extending the COLLAGE plan structure to incorporate an embedded "justification" for each plan action, relation, and binding. This justification structure will serve as a framework for tracking and correcting constraint violations. Search through a sequence of "reformations" must obey the following rules: (1) it cannot backtrack over nodes associated with portions of the plan that have already been executed; and (2) it cannot backtrack to previous reformations.

9.2 User-Planner Integration

In our experience with coordination-intensive domains, we have come to recognize the importance of *user-planner integration.* If users have deep knowledge of a domain and a vested interest in the form of the final plan, they will not willingly utilize a planner unless it allows for their direct input into the planning process. Unfortunately, the planning community has largely ignored this problem. Our attempt to deal with user-planner integration has resulted in the development of COLLIE, the **Collage Interface Environment**. A COLLIE user can visualize the growing plan, inspect features of each action, relation, and binding, and understand the relationship between plan structure and domain constraints. Tools are provided for viewing a graphical representation of the domain localization structure, visualizing the localized search process, and editing the domain description and knowledge base. Tracing and stepping options are provided for monitoring planning and execution. Ultimately, we will allow the user to modify the plan itself and to interact more directly with the constraint activation and search control mechanism.

9.3 Localization Studies

Another goal of the COLLAGE project is to deepen our understanding of localized search. We are embarking upon an empirical study to test a variety of search strategies over a suite of problems from the office building domain. The office building problems will vary in several dimensions:

- *The amount of backtracking required.* This will be varied by using resource limitations.
- *Constraint algorithm difficulty*, varied by using alternative constraint forms.
- *Domain localization structure.* In particular, we are developing an automatic localization generator, LOC, described below.
- *Office building structure and size.* Our encoding of office buildings in terms of "pods" allows us to easily and automatically generate new building descriptions.

Using the region agenda mechanism and heuristics that can be associated with choice nodes, we will also be experimenting with alternative search strategies. Another interesting test will be to vary the amount of temporal closure and consistency maintenance that is performed. For example, the default temporal closure strategy in COLLAGE is to perform closure each time a temporal relation is added. An alternative, more relaxed, approach would be to perform closure only after a fix is completed or at the end of each incarnation.

In order to systematically generate possible localizations for a domain description, we have developed a COLLAGE localization generator, LOC, that searches through a "localization space." Each node in this space is associated with a localization, and each arc is associated with a localization "transform" that transforms one localization into another. Currently, LOC generates new localizations by merging regions together – i.e. by decreasing the level of partitioning for a domain. The root node of the LOC space is associated with a highly partitioned localization: each action type is associated with its own region and each constraint is associated with a region that includes relevant "action-type" subregions. Transforms either collapse a set of regions together or restructure the region topology. Eight distinct transforms that have been identified and implemented.

The overall goal of LOC is to remove regional overlap while still retaining as many localization benefits as possible. Since the current transforms only increase constraint scope, they can only affect search cost; they do not affect plan correctness. In the future, we also plan to incorporate transforms that split a region up into multiple regions. For instance, empirical testing could be used to determine the true scope of specific constraints and thereby provide information for further partitioning of the domain region structure. As a by-product of our work with LOC and the office building domain test suite, we also hope to come up with a localization *learning* approach that automatically discovers domain-dependent and domain-independent localization heuristics.

10 Conclusion

This paper has presented COLLAGE, a planner that utilizes a diverse suite of action-based plan construction methods within a localized search framework. This unique approach to domain representation and planning was motivated by the requirements of coordination-intensive domains. Expressive and natural to use, COLLAGE's action-based constraints provide a substrate for cost-effective planning. In our experience, with realistic domains, the action-based approach has obviated the need for planning methods based on the modal truth criterion. Moreover, in both analytical and empirical studies, COLLAGE's use of localized search has yielded significant planning cost reduction, while still allowing for subproblem interactions. This paper has also tried to demonstrate how a planning "ontology" based on a six-dimensional view of the planning process can serve as an excellent framework for expanding our view of what planning *is*, and for understanding and comparing the myriad planning techniques that have been developed.

Acknowledgements

First and foremost, I would like to acknowledge the members of the COLLAGE project. Andrew Philpot and Lise Getoor, for their work on the development of COLLAGE, for gathering the empirical results presented in this paper, and for their comments on this paper. Anna Karlin provided invaluable assistance with the localization complexity analysis. John Allen and John Bresina also provided useful input into this paper. Others who have provided advice and encouragement over the past few years include Mark Drummond, Peter Friedland, Rao Kambhampati, Rich Keller, Steve Minton, Steven Rubin, and Monte Zweben.

References

[1] Allen, J.F. "An Interval-Based Representation of Temporal Knowledge," *Proceedings of the Seventh International Joint Conference on Artificial Intelligence (IJCAI-81)*. Vancouver, B.C. Canada (August 1981).

[2] Chapman, D. "Planning for Conjunctive Goals," *Artificial Intelligence*, Volume 32, pp. 333-377 (1987).

[3] Christensen, J. "A Hierarchical Planner that Generates Its Own Hierarchies," in *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI90)*, Boston, Massachusetts, pp. 1004-1009 (1990).

[4] Currie, K. and A. Tate, "O-Plan: The Open Planning Architecture," *Artificial Intelligence*, Volume 52, pp. 49-86 (1991).

[5] Drummond, M. and J. Bresina, "Anytime Synthetic Projection: Maximizing the Probability of Goal Satisfaction," *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90)*, pp. 138–144 (1990).

[6] Etzioni, O. and N. Lesh, "Planning with Incomplete Information in the UNIX Domain," *Proceedings of the 1993 AAAI Spring Symposium on Foundations of Automatic Planning: The Classical Approach and Beyond*, Stanford, California (1993).

[7] Fikes, R.E, Hart, P.E., and Nilsson, N.J. "Learning and Executing Generalized Robot Plans," *Artificial Intelligence*, Volume 3, Number 4, pp. 251-288, (1972).

[8] Firby, R.J. *Adaptive Execution in Complex Dynamic Worlds*, Yale University Computer Science Department, TR 672 (1989).

[9] Georgeff, M.P. and A.L. Lansky, "Reactive Reasoning and Planning," *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, Seattle, Washington (July 1987).

[10] Hammond, K.J. "CHEF: A Model of Case-Based Planning," *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, pp. 261–271 (1986).

[11] Kaelbling, L. "An Architecture for Intelligent Reactive Systems," in *Reasoning About Actions and Plans*, M.Georgeff and A.Lansky (editors), Morgan Kaufmann (1987).

[12] Kambhampati, S. and J. Hendler "A Validation-Structure-Based Theory of Plan Modification and Reuse," *Artificial Intelligence*, Volume 55, Numbers 2-3, pp. 193–259 (1992).

[13] Khartam, N.A. *Investigating the Utility of Artificial Intelligence Techniques for Automatic Generation of Construction Project Plans*, Doctoral Dissertation, Stanford University Department of Civil Engineering (1989).

[14] Knoblock, C.A. "Learning Abstraction Hierarchies for Problem Solving," in *Seventh International Workshop on Machine Learning*, pp. 923-928 (1990).

[15] Knoblock, C.A., Tenenberg, J.D., and Q. Yang. "A Spectrum of Abstraction Hierarchies for Planning," *Proceedings of the 1990 Workshop on Automatic Generation of Approximations and Abstractions*, Boston, Massachusetts, pp.24–35 (1990).

[16] Korf, R.E. "Planning as Search: A Quantitative Approach," *Artificial Intelligence (33.1)*, pp. 65-88 (1987).

[17] Lansky, A.L. and S.S. Owicki, "GEM: A Tool for Concurrency Specification and Verification," *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing (PODC-83)*, pp. 198–212 (1983).

[18] Lansky, A.L. "A Representation of Parallel Activity Based on Events, Structure, and Causality," in *Reasoning About Actions and Plans*, M. Georgeff and A. Lansky (editors), Morgan Kaufmann, pp. 123-160 (1987).

[19] Lansky, A.L. "Localized Event-Based Reasoning for Multiagent Domains," *Computational Intelligence Journal, Special Issue on Planning (4,4)* (1988).

[20] Lansky, A.L. "Localized Representation and Planning," in *Readings in Planning*, J. Allen, J. Hendler, and A. Tate (editors), Morgan Kaufmann (1990).

[21] Lansky, A.L. "Localized Search for Multiagent Domains," *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*, Sydney, Australia, pp. 252–258 (1991).

[22] Lansky, A.L. "Localization vs. Abstraction: A Comparison of Two Search Reduction Techniques," *Proceedings of the AAAI-92 Workshop on Approximation and Abstraction of Computational Theories.*

[23] Lansky, A.L. and A.G. Philpot, "AI-Based Planning for Data Analysis Tasks," *Proceedings of the Ninth IEEE Conference on Artificial Intelligence for Applications (CAIA-93)*, Orlando, Florida (1993).

[24] Lansky, A.L. and A.G. Philpot, "COLLAGE: A Diversified Constraint-Based Planning Architecture," *Proceedings of the 1993 AAAI Spring Sympoisum on Foundations of Automatic Planning: The Classical Approach and Beyond*, Stanford, California (1993).

[25] Mackworth, A. K. "Consistency in Networks of Relations," *Artificial Intelligence*, Volume 8, pp. 99-118 (1977).

[26] Maes, P. "The Dynamics of Action Selection," *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, Detroit, Michigan, pp. 991–997 (1989).

[27] McAllester, D. and D. Rosenblitt, "Systematic Nonlinear Planning," *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)*, pp. 634–639 (1991).

[28] Missiaen, L. *Localized Abductive Planning with the Event Calculus*, PhD. Thesis, Department of Computer Science, K.U. Leuven, Belgium (1991).

[29] Penberthy, J.S. and D.S. Weld, "Temporal Planning with Constraints," *Proceedings of the 1993 AAAI Spring Sympoisum on Foundations of Automatic Planning: The Classical Approach and Beyond*, Stanford, California (1993).

[30] Sacerdoti, E.D. *A Structure for Plans and Behavior*, Elsevier North-Holland, Inc., New York, New York (1977).

[31] Schoppers, M. "Universal Plans for Reactive Robots in Unpredictable Environments," *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87)*, Milan, Italy, pp. 1039-1046 (1987).

[32] Tate, A. "Goal Structure, Holding Periods, and 'Clouds'," in *Reasoning About Actions and Plans*, M.Georgeff and A.Lansky (editors), Morgan Kaufman Publishers, pp. 267-277 (1987).

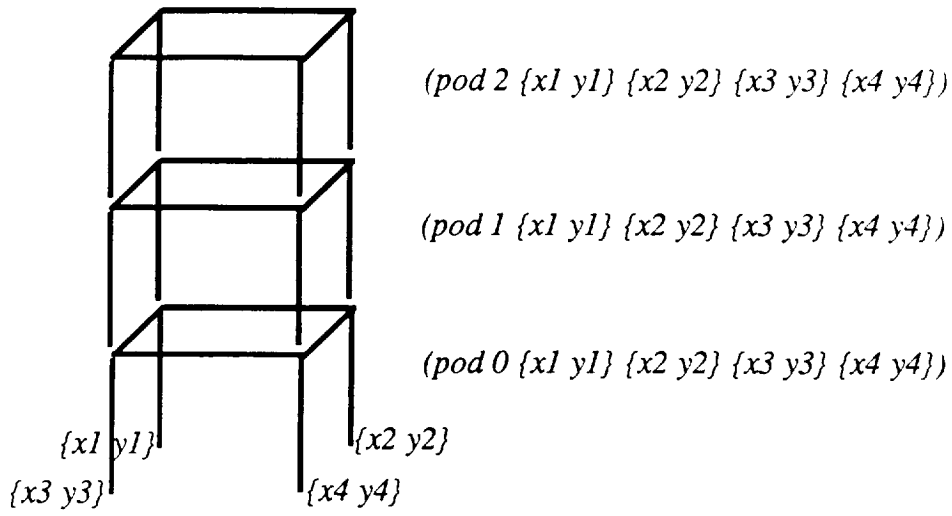
[33] Veloso, M. "Planning for Complex Tasks: Replay and Merging of Multiple Plans," *Proceedings of the 1993 AAAI Spring Sympoisum on Foundations of Automatic Planning: The Classical Approach and Beyond*, Stanford, California (1993).

[34] Wellman, M.P. "Challenges of Decision Theoretic Planning," *Proceedings of the 1993 AAAI Spring Sympoisum on Foundations of Automatic Planning: The Classical Approach and Beyond*, Stanford, California (1993).

[35] Wilkins, D. *Practical Planning*, Morgan Kaufmann Publishers, San Mateo, California (1989).

APPENDIX

The following provides the region type definitions and region instances for the office building domain (for the localization that displayed the best results in our empirical studies). This domain description is followed by a fact data base for a particular problem instance - an L-shaped office building with a basement and four finished floors.



As described earlier, each office building is described in terms of a set of *pods*. Each pod may be viewed as a cube-like building-block (see above). A pod definition contains a floor number and four coordinate points. The presence of a particular pod in the fact data base will result in the addition of four columns (for the specified floor, at the four specified coordinates), four walls (on that floor, between the four coordinates), four beams connecting the tops of the four columns, and a deck laid on top of the beams. Pods are "stacked" one on top of the other, with the deck of the top pod forming the "roof" and the beams of the bottom bod forming the foundational piers.

Region Type Definitions

```
;; BUILD BEAMS

(def-region-type all-beams-type
 :action-type (build-beam floor coord coord)

 :constraint
 (action
  :condition ((fact (pod ?f ?c1 ?c2 ?c3 ?c4)))
  :actions ((build-beam ?f ?c1 ?c2)
            (build-beam ?f ?c3 ?c4)
            (build-beam ?f ?c1 ?c3)
            (build-beam ?f ?c2 ?c4))))
```

```
;; BUILD COLUMNS

(def-region-type all-columns-type
 :action-type (build-column floor coord)

 :constraint
 (action
  :condition ((fact (pod ?f ?c1 ?c2 ?c3 ?c4)))
  :actions ((build-column ?f ?c1)
            (build-column ?f ?c2)
            (build-column ?f ?c3)
            (build-column ?f ?c4))))


;; BUILD DECKS

(def-region-type all-decks-type
 :action-type (build-deck floor coord coord coord coord)

 :constraint
 (action
  :condition ((fact (pod ?f ?c1 ?c2 ?c3 ?c4)))
  :actions ((build-deck ?f ?c1 ?c2 ?c3 ?c4))))


;; BUILD WALLS -- none at the basement level

(def-region-type all-walls-type
 :action-type (build-wall floor coord coord)

 :constraint
 (action
  :condition ((fact (pod ?f ?c1 ?c2 ?c3 ?c4))
              (test (> ?f 0)))
  :actions ((build-wall ?f ?c1 ?c2)
            (build-wall ?f ?c3 ?c4)
            (build-wall ?f ?c1 ?c3)
            (build-wall ?f ?c2 ?c4))))


;; BASEMENT AND FOUNDATION

(def-region-type groundlevel-type
 :action-type (build-footing coord)

 :constraint
 (action
  :condition ((fact (pod 0 ?c1 ?c2 ?c3 ?c4)))
  :actions ((build-footing ?c1)
            (build-footing ?c2)
            (build-footing ?c3)
            (build-footing ?c4)))

 :constraint
 (all-match-precede
  :actions ((build-footing ?c) (build-column 0 ?c))))
```

```
;;; CONSTRAINTS BETWEEN BASIC BUILDING COMPONENTS


(def-region-type column-beam-nexus-type
 :constraint
 (tempbefore
  :actions ((build-column ?f ?c1) (build-beam ?f ?c1 ?c2)))

 :constraint
 (tempbefore
  :actions ((build-column ?f ?c2) (build-beam ?f ?c1 ?c2))))



(def-region-type beam-deck-nexus-type
 :constraint
 (tempbefore
  :actions ((build-beam ?f ?c1 ?c2) (build-deck ?f ?c1 ?c2 ?c3 ?c4)))

 :constraint
 (tempbefore
  :actions ((build-beam ?f ?c3 ?c4) (build-deck ?f ?c1 ?c2 ?c3 ?c4)))

 :constraint
 (tempbefore
  :actions ((build-beam ?f ?c1 ?c3) (build-deck ?f ?c1 ?c2 ?c3 ?c4)))

 :constraint
 (tempbefore
  :actions ((build-beam ?f ?c2 ?c4) (build-deck ?f ?c1 ?c2 ?c3 ?c4))))



(def-region-type deck-column-nexus-type
 :constraint
 (tempafter
  :condition ((action (build-column ?f ?c1))
              (test (> ?f 0))
              (make (?belowf (- ?f 1))))
  :actions ((build-deck ?belowf ?c1 ?c2 ?c3 ?c4) (build-column ?f ?c1)))

 :constraint
 (tempafter
  :condition ((action (build-column ?f ?c2))
              (test (> ?f 0))
              (make (?belowf (- ?f 1))))
  :actions ((build-deck ?belowf ?c1 ?c2 ?c3 ?c4) (build-column ?f ?c2)))

 :constraint
 (tempafter
  :condition ((action (build-column ?f ?c3))
              (test (> ?f 0))
              (make (?belowf (- ?f 1))))
  :actions ((build-deck ?belowf ?c1 ?c2 ?c3 ?c4) (build-column ?f ?c3)))
```

```
 :constraint
 (tempafter
  :condition ((action (build-column ?f ?c4))
              (test (> ?f 0))
              (make (?belowf (- ?f 1))))
  :actions ((build-deck ?belowf ?c1 ?c2 ?c3 ?c4) (build-column ?f ?c4))))


(def-region-type beam-wall-nexus-type
 :constraint
 (tempbefore
  :actions ((build-beam ?f ?c1 ?c2) (build-wall ?f ?c1 ?c2))))



;;; FINISHING A FLOOR LEVEL

(def-region-type all-floors-type
 :action-type (do-finish-floor floor)
 :action-type (dummy-first-finish-floor floor)
 :action-type (dummy-last-finish-floor floor)

 :constraint
 (action
  :condition ((fact (floor ?f))
              (test (> ?f 0)))
  :actions ((do-finish-floor ?f)))

 :constraint
 (decompose
  :action ((do-finish-floor ?f))
  :decompositions
  ((:subactions
    (#1=(dummy-first-finish-floor ?f)
     #2=(do-partitioning ?f)
     #3=(do-ceiling ?f)
     #4=(do-flooring ?f)
     #5=(dummy-last-finish-floor ?f))
    :first-subaction #1#
    :last-subaction #5#
    :relations ((:before #1# #2#)
                (:before #1# #3#)
                (:before #1# #4#)
                (:before #2# #5#)
                (:before #3# #5#)
                (:before #4# #5#)))))

 :constraint
 (all-match-precede
  :actions ((do-drywall ?f) (do-ceiling-grid ?f)))

 :constraint
 (all-match-precede
  :actions ((suspended-ceiling ?f) (finish-flooring ?f)))
```

```
 :constraint
 (all-match-precede
  :actions ((painting ?f) (finish-flooring ?f)))

 :constraint
 (all-match-precede
  :actions ((build-deck ?f ?c1 ?c2 ?c3 ?c4) (do-finish-floor ?f)))

 :constraint
 (all-match-precede
  :condition ((action (build-wall ?f ?c1 ?c2))
              (fact (external-wall ?f ?c1 ?c2)))
  :actions ((build-wall ?f ?c1 ?c2) (do-finish-floor ?f))))


;;; PARTITIONING A FLOOR LEVEL

(def-region-type partitioning-type
 :action-type (do-partitioning floor)
 :action-type (m-and-e-wall-services floor)
 :action-type (drywall-studs floor)
 :action-type (taping floor)
 :action-type (painting floor)
 :action-type (wall-fixtures floor)
 :action-type (door-frames floor)
 :action-type (doors floor)
 :action-type (window-frames floor)
 :action-type (glazing floor)
 :action-type (do-drywall floor)
 :action-type (start-drywall floor)
 :action-type (finish-drywall floor)

 :constraint
 (decompose
  :action ((do-partitioning ?f))
  :decompositions
  ((:subactions (#1=(m-and-e-wall-services ?f)
                 #2=(drywall-studs ?f)
                 #3=(do-drywall ?f)
                 #4=(taping ?f)
                 #5=(painting ?f)
                 #6=(wall-fixtures ?f)
                 #7=(door-frames ?f)
                 #8=(doors ?f)
                 #9=(window-frames ?f)
                 #10=(glazing ?f))
    :first-events (#1# #2# #7# #9#)
    :last-events (#6# #8# #10#)
    :relations ((:before #1# #4#)
                (:before #2# #3#)
                (:before #2# #7#)
                (:before #3# #4#)
                (:before #4# #5#)
                (:before #5# #6#)
                (:before #7# #8#)
                (:before #9# #10#)))))
```

```
 :constraint
 (decompose
  :action ((do-drywall ?f_floor))
  :decompositions
  ((:subactions (#1=(start-drywall ?f)
                 #2=(finish-drywall ?f))
    :first-event #1#
    :last-event #2#
    :relations ((:before #1# #2#)))))

 :constraint
 (tempbefore
  :actions ((m-and-e-wall-services ?f) (finish-drywall ?f))))


;; CEILING FOR A FLOOR LEVEL

(def-region-type ceiling-type
 :action-type (do-ceiling floor)
 :action-type (m-and-e-ceiling-services floor)
 :action-type (suspended-ceiling floor)
 :action-type (ceiling-fixtures floor)
 :action-type (do-ceiling-grid floor)
 :action-type (start-ceiling-grid floor)
 :action-type (finish-ceiling-grid floor)

 :constraint
 (decompose
  :action ((do-ceiling ?f))
  :decompositions
  ((:subactions (#1=(m-and-e-ceiling-services ?f)
                 #2=(do-ceiling-grid ?f)
                 #3=(suspended-ceiling ?f)
                 #4=(ceiling-fixtures ?f))
    :first-events (#1# #2#)
    :last-event #4#
    :relations ((:before #1# #3#)
                (:before #2# #3#)
                (:before #3# #4#)))))

 :constraint
 (decompose
  :action ((do-ceiling-grid ?f))
  :decompositions
  ((:subactions (#1=(start-ceiling-grid ?f)
                 #2=(finish-ceiling-grid ?f))
    :first-event #1#
    :last-event #2#
    :relations ((:before #1# #2#)))))

 :constraint
 (tempbefore
  :actions ((m-and-e-ceiling-services ?f)
            (finish-ceiling-grid ?f))))
```

```
;; FLOORING A FLOOR LEVEL

(def-region-type flooring-type
 :action-type (do-flooring floor)
 :action-type (start-flooring floor)
 :action-type (lay-carpet floor)
 :action-type (finish-flooring floor)

 :constraint
 (decompose
  :action ((do-flooring ?f))
  :decompositions
  ((:subactions (#1=(start-flooring ?f)
                 #2=(lay-carpet ?f)
                 #3=(finish-flooring ?f))
    :first-event #1#
    :last-event #3#
    :relations ((:before #1# #2#)
                (:before #2# #3#))))))
```

Region Instances

```
(defregion (all-beams all-beams-type))

(defregion (all-columns all-columns-type))

(defregion (all-decks all-decks-type))

(defregion (all-walls all-walls-type))

(defregion (groundlevel groundlevel-type)
  :subregion all-columns)

(defregion (column-beam-nexus column-beam-nexus-type)
  :subregion all-beams
  :subregion all-columns)

(defregion (beam-deck-nexus beam-deck-nexus-type)
  :subregion all-beams
  :subregion all-decks)

(defregion (beam-wall-nexus beam-wall-nexus-type)
 :subregion all-beams
 :subregion all-walls)

(defregion (deck-column-nexus deck-column-nexus-type)
  :subregion all-decks
  :subregion all-columns)
```

```
(defregion (all-floors all-floors-type)
 :subregion all-decks
 :subregion all-walls
 :subregion (:generate (partitioning partitioning-type)
             :limit :infinity)
 :subregion (:generate (ceiling ceiling-type)
             :limit :infinity)
 :subregion (:generate (flooring flooring-type)
             :limit :infinity))
```

Fact base for office building with a basement and four finished floors:

```
(deffact (floor 0))
(deffact (floor 1))
(deffact (floor 2))
(deffact (floor 3))
(deffact (floor 4))

(deffact (pod 0 {0 1} {1 1} {0 0} {1 0}))
(deffact (pod 0 {1 1} {2 1} {1 0} {2 0}))
(deffact (pod 0 {0 2} {1 2} {0 1} {1 1}))
(deffact (pod 0 {0 3} {1 3} {0 2} {1 2}))

(deffact (pod 1 {0 1} {1 1} {0 0} {1 0}))
(deffact (pod 1 {1 1} {2 1} {1 0} {2 0}))
(deffact (pod 1 {0 2} {1 2} {0 1} {1 1}))
(deffact (pod 1 {0 3} {1 3} {0 2} {1 2}))

(deffact (pod 2 {0 1} {1 1} {0 0} {1 0}))
(deffact (pod 2 {1 1} {2 1} {1 0} {2 0}))
(deffact (pod 2 {0 2} {1 2} {0 1} {1 1}))
(deffact (pod 2 {0 3} {1 3} {0 2} {1 2}))

(deffact (pod 3 {0 1} {1 1} {0 0} {1 0}))
(deffact (pod 3 {1 1} {2 1} {1 0} {2 0}))
(deffact (pod 3 {0 2} {1 2} {0 1} {1 1}))
(deffact (pod 3 {0 3} {1 3} {0 2} {1 2}))

(deffact (pod 4 {0 1} {1 1} {0 0} {1 0}))
(deffact (pod 4 {1 1} {2 1} {1 0} {2 0}))
(deffact (pod 4 {0 2} {1 2} {0 1} {1 1}))
(deffact (pod 4 {0 3} {1 3} {0 2} {1 2}))

(deffact (external-wall 1 {0 1} {0 0}))
(deffact (external-wall 1 {0 0} {1 0}))
(deffact (external-wall 1 {1 0} {2 0}))
(deffact (external-wall 1 {2 1} {2 0}))
(deffact (external-wall 1 {1 1} {2 1}))
(deffact (external-wall 1 {0 2} {0 1}))
(deffact (external-wall 1 {1 2} {1 1}))
(deffact (external-wall 1 {0 3} {0 2}))
(deffact (external-wall 1 {1 3} {1 2}))
(deffact (external-wall 1 {0 3} {1 3}))
```

```
(deffact (external-wall 2 {0 1} {0 0}))
(deffact (external-wall 2 {0 0} {1 0}))
(deffact (external-wall 2 {1 0} {2 0}))
(deffact (external-wall 2 {2 1} {2 0}))
(deffact (external-wall 2 {1 1} {2 1}))
(deffact (external-wall 2 {0 2} {0 1}))
(deffact (external-wall 2 {1 2} {1 1}))
(deffact (external-wall 2 {0 3} {0 2}))
(deffact (external-wall 2 {1 3} {1 2}))
(deffact (external-wall 2 {0 3} {1 3}))

(deffact (external-wall 3 {0 1} {0 0}))
(deffact (external-wall 3 {0 0} {1 0}))
(deffact (external-wall 3 {1 0} {2 0}))
(deffact (external-wall 3 {2 1} {2 0}))
(deffact (external-wall 3 {1 1} {2 1}))
(deffact (external-wall 3 {0 2} {0 1}))
(deffact (external-wall 3 {1 2} {1 1}))
(deffact (external-wall 3 {0 3} {0 2}))
(deffact (external-wall 3 {1 3} {1 2}))
(deffact (external-wall 3 {0 3} {1 3}))

(deffact (external-wall 4 {0 1} {0 0}))
(deffact (external-wall 4 {0 0} {1 0}))
(deffact (external-wall 4 {1 0} {2 0}))
(deffact (external-wall 4 {2 1} {2 0}))
(deffact (external-wall 4 {1 1} {2 1}))
(deffact (external-wall 4 {0 2} {0 1}))
(deffact (external-wall 4 {1 2} {1 1}))
(deffact (external-wall 4 {0 3} {0 2}))
(deffact (external-wall 4 {1 3} {1 2}))
(deffact (external-wall 4 {0 3} {1 3}))
```